Actes des 2èmes journées sur l'Ingénierie Dirigée

Laurence Duchien, Cedric Dumoulin

IDM 6

http://planetmde.org/idm06

2e journées sur l'

Ingénierie Dirigée par les Modèles

Lille 26-28 juin 2006

ceal list · INRIA · IRCICA · LIFL · Polytech'Lille · USTL · planetmde.org

Actes des 2èmes journées sur l'Ingénierie Dirigée par les Modèles

Editeurs : Laurence Duchien et Cédric Dumoulin

Ce document est téléchargeable à partir du site de la conférence IDM06:
http://planetmde.org/idm06

ISBN 2-7261-1290-8
Lille, Juin 2006

IDM 06

Actes des 2èmes journées sur l'Ingénierie Dirigée par les Modèles

Lille, 26-28 Juin 2006

Préface

Ces dernières années, les nouveaux logiciels ont gagné en complexité, essentiellement parce qu'ils doivent fonctionner dans des environnements contraints ou changeants. Dans le domaine des applications embarquées, autogérées ou encore nomades, lorsqu'un nouveau logiciel est conçu, le développeur va le plus souvent fonder son développement sur la réutilisation d'éléments existants, mais aussi travailler à partir d'une spécification qui peut changer dans le temps et dans l'espace. Dans le monde actuel du génie logiciel, au-delà de la réutilisation, ces changements continus ou encore l'évolution dans le temps sont devenus des normes. L'issue réaliste est alors de produire des logiciels de façon automatique en permettant la gestion des changements temporels et spatiaux pour prendre en compte les versions successives et les variantes de ces logiciels.

L'ingénierie dirigée par les modèles (IDM), appelée en anglais MDE (Model-Driven Engineering) ou aussi MDD (Model-Driven Development) place le modèle au centre du processus de conception et permet à cette notion de modèle de passer d'un rôle contemplatif à un rôle unificateur vis-à-vis des autres activités du cycle de développement du logiciel. L'IDM doit alors être vu non pas comme une révolution, mais comme un moyen d'intégrationde différents espaces techniques pour aller vers une production automatisée des logiciels.

L'ingénierie dirigée par les modèles apporte alors des solutions à la construction de ces nouveaux logiciels en proposant des approches de modélisation, de métamodélisation, de détermination du domaine, de transformation et de prise en compte des plates-formes. Ces approches sont accompagnées de démarches de conception et de moyens de génération de code, mais également de validation et de vérification de la conformité des modèles produits vis-à-vis des métamodèles. Elles sont proches des idées actuelles comme la programmation générative, les langages spécifiques de domaine (DSL), le MIC (Model Integrating Computing) ou encore les usines à logiciels (Software factories).

Après le succès des journées IDM à Paris en 2005, la seconde édition de ces journées se déroule à Lille et a pour objectif de rassembler les chercheurs francophones intéressés par ce domaine et souhaitant participer à la structuration de cette communauté scientifique émergente.

Ces journées sont multiformes et accueillent deux tutoriaux, l'un sur les concepts fondamentaux de l'IDM et l'autre sur le développement coordonné entre UML 2 et les Enterprise Java Beans, onze sélectionnés et répartis en cinq sessions, une session poster permettant l'expression d'idées nouvelles non encore abouties, un atelier de travail sur les motifs de métamodélisation, une assemblée générale de l'action IDM qui est tranversale aux différents GDR CNRS et deux démonstrations de plates-formes : Kermeta et Obeo Agility.

Nous vous souhaitons à tous un bon IDM 2006 à Lille.

Les éditeurs,
Laurence Duchien, Cédric Dumoulin

Actes de la conférence

Tutorial 1

Session 1

Session 2

Demonstration d'outils

Tutorial 2

Session 3

Session 4

Session 5

Actes de l'atelier de travail *Motifs de méta-modélisation*

Actes de la conférence

Concepts fondamentaux de l'IDM

De l'Ancienne Egypte à l'Ingénierie des Langages

Tutorial

Jean-Marie Favre
Archéologue des langages logiciels
Université Joseph Fourier
Grenoble, France
http://www-adele.imag.fr/~jmfavre

métamodèle ougaritique cunéiforme

Au-delà du standard MDA de l'OMG, l'Ingénierie Dirigée par les Modèles (IDM) est devenue en quelques années un thème de discussion chez les informaticiens, aussi bien dans le monde académique que dans l'industrie. Le nombre de conférences et d'ateliers portant sur ce thème est en pleine expansion comme en témoigne par exemple la section "événements" de http://planetmde.org. Les discussions portant sur l'IDM mènent souvent à des polémiques entre détracteurs et prophètes visionnaires.

Ponctuées d'acronymes (MDA, CWM, PIM, HUTN, MOF, CIM, EXPRESS, SDAI, MDR, EMOF, JMI, EMF, ECORE, QVT, SPEM, XMI, pour n'en citer que quelques uns), mais aussi de standards aussi variés que complexes à maîtriser, le ticket d'entrée dans la jungle de l'IDM est très élevé. Derrière ce foisonnement de technologies plus où moins obscures, et cette terminologie qui semble ésotérique, certains se demandent d'ailleurs si la communauté du MDA n'est pas en train de tenter de "ré-inventer la roue", sans même peut être s'en apercevoir.

Ce qui avant s'appelait un "fichier", s'appelle désormais un "modèle" ; une grammaire s'appelle désormais "métamodèle", etc. En fait le jargon méta-tralala n'est pas fait pour arranger les choses. Il a même plutôt tendance à jouer le rôle de repoussoir. Déjà que le monde de la modélisation et ses dessins "à la UML" n'avait pas nécessairement bonne presse dans le monde des "vrais" développeurs, rajouter des niveaux d'abstractions semble le meilleur moyen de se déconnecter définitivement de la réalité informatique, c'est-à-dire du code.

Pour aggraver encore plus la situation, le monde de l'IDM est sous actuellement l'emprise de luttes de clans plus ou moins faciles à identifier depuis l'extérieur : aux pro-UML-exécutable s'affrontent les pro-DSLs-à-chacun-son-langage, eux même en opposition aux pro-UML-à-tout-faire. Et ce n'est là sans parler des problèmes byzantins "d'alignements" de langages aux multiples versions plus ou moins incompatibles. Bref, s'approprier l'IDM consiste a et essayer tant bien que mal à comprendre la raison d'être de ces technologies et langages instables, mais aussi la complémentarité entre les certitudes assénées par chaque clan. Face à cette complexité accidentelle et à ces positions quelques fois dogmatiques, ce tutorial vise à déterminer ce qu'est l'essentiel. Dans ce tutorial nous cultivons le doute ; car comme le dit Descartes c'est le doute qui mène à la vérité. Qu'a-t-on inventé depuis l'invention de la roue? Le MDA ?

L'objectif de ce tutorial est d'étudier les concepts fondamentaux de l'IDM en montrant leurs incarnations concrètes non seulement dans les approches actuelles mais aussi leur rôle dans l'histoire de l'humanité. Ces concepts fondamentaux seront représentés sous la forme d'un … "mégamodèle". Derrière ce terme, qui n'a finalement peu d'importance, en repartant de l'invention de la roue, en traversant les millénaires, ce tutorial montre comment l'on arrive progressivement aux notions fondamentales de l'IDM. Par exemple les notions de "modèle", de "langage", de "métamodèle", de "mégamodèle", ainsi que les problèmes associés se retrouvent dés l'antiquité.

Par exemple la sculpture présentée ci-dessous modélise la relation entre la notion de modèle et de langage en respectant la disposition que l'on retrouve dans les pyramides. Le dieu Thot, inventeur du langage selon la mythologie égyptienne, aurait fait le don de métamodèles aux hommes. Ce n'est pourtant là qu'un mythe. Par contre le lecteur trouvera au sommet de la page précédente l'un des premiers métamodèles ougaritiques créé par les hommes sous la forme d'une tablette cunéiforme. En fait comme nous le verons dans ce tutorial cette tablette d'argile est l'un des témoins d'une étape majeure dans l'histoire de l'humanité, le point de départ de la démocratisation de la modélisation.

Mégamodèle égyptien représentant le dieu Thot et le scribe Imothept. Musée du Louvre.

Tout au long des derniers millénaires on retrouve donc les concepts fondamentaux de l'IDM. Ce n'est finalement pas surprenant, car lorsque des concepts sont réellement *fondamentaux*, il est quelque peu présomptueux au regard de l'histoire de l'humanité d'imaginer qu'ils viennent d'être "inventés". Il nous semble ainsi difficile de défendre la position selon laquelle, la nouveauté de l'Ingénierie Dirigée par les Modèles serait l'utilisation de modèles. Au contraire, l'une des nouveautés, est la matérialisation systématique des langages de modélisation utilisés, et ce sous la forme de métamodèle. On devrait ainsi parler plus justement d'Ingénierie Dirigée par les Métamodèles.

Au-delà du MDA, et de l'Ingénierie Dirigée par les Modèles, nous pensons plus globalement que l'informatique s'oriente à plus ou moins long terme vers l'Ingénierie des Langages. Différents indices étayant cette thèse seront présentés en montrant autant des exemples tirés de l'histoire de l'écriture, de la cartographie ou encore de l'art conceptuel que des différents espaces techniques actuels (Modelware, Grammarware, XMLware, Dataware, Ontologyware, …). Bien que cette approche puissent surprendre au départ, les concepts fondamentaux de l'IDM sont si généraux et stables qu'il n'est finalement pas étonnant de les retrouver dans d'autres domaines, et ce depuis l'antiquité. Par exemple l'histoire de l'écriture montre une cohabitation pacifique entre langages spécifiques de domaines et langages généralistes. Considérons aussi par exemple l'héraldique, les partitions de musique, les numérations, les calendriers, les transcriptions phonétiques, etc. Dans la plupart des domaines ont retrouve, sous une forme ou sous une autre, les notions d'évolution des langages, d'adaptation de langages, de composition et d'extension de langages, etc. La notion de style ou de profile existait bien avant UML.

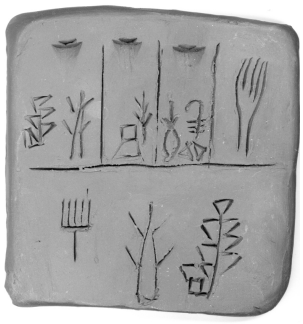
En partant de multiples exemples et en le mettant en relation avec les problématiques concrètes auxquelles les monde de l'IDM doit faire fasse, ce tutorial esquissera les contours de ce qui pourrait devenir un nouveau thème de recherche que nous appelons "linguistique logicielle", c'est-à-dire l'étude des langages logiciels et de leur évolution dans le temps et dans l'espace. Les différentes branches de la linguistique, traditionnellement appliquées aux langages dites naturelles, peuvent en fait considérées dans le contexte des langages logiciels. C'est le car par exemple de la socio linguistique, de la politique linguistique, de le l'étude de l'acquisition des langages, de leur usage, etc. Notons que l'on fait ici référence à la notion de langage qu'il soit décrit aussi bien sous forme de grammaires, de métamodèles, d'ontologies, de DSLs, de schémas, ou encore sous toute autre forme.

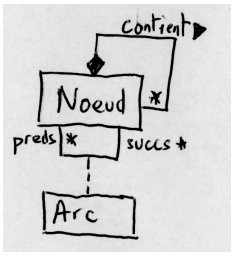
Une telle vision unificatrice permet non seulement de rassembler ou de confronter des problématiques étudiées par des communautés différentes en informatique (Génie Logiciel, Web sémantique, Ingénierie des systèmes interactifs, Ingénierie des systèmes d'informations, etc.), mais aussi de les relier à des disciplines plus anciennes telle que la sémiotique, la linguistique, etc. Nous montrerons que la démarche poursuivie dans ce tutorial est directement compatible d'une part avec l'évolution technologique actuelle, mais aussi avec les racines même de l'informatique, qui rappelons le, proviennent en partie de la linguistique. L'héritage de Thot est bel et bien toujours présent, et les hommes sont encore loin d'en avoir fait le tour. Et ce n'est certainement ni Champollion, ni Kosuth, ni Chomsky qui nous contrediront…

Modèle égyptien. Musée de Grenoble.

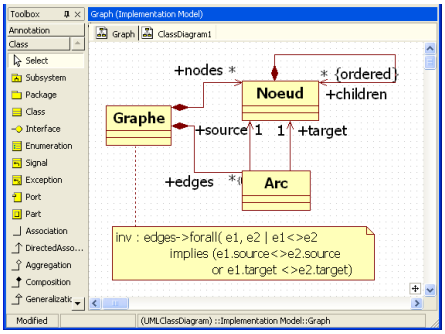
Le tutorial sera ponctué de différentes "énigmes" qu'il s'agira de résoudre. Voici ci-dessous quelques exemples.

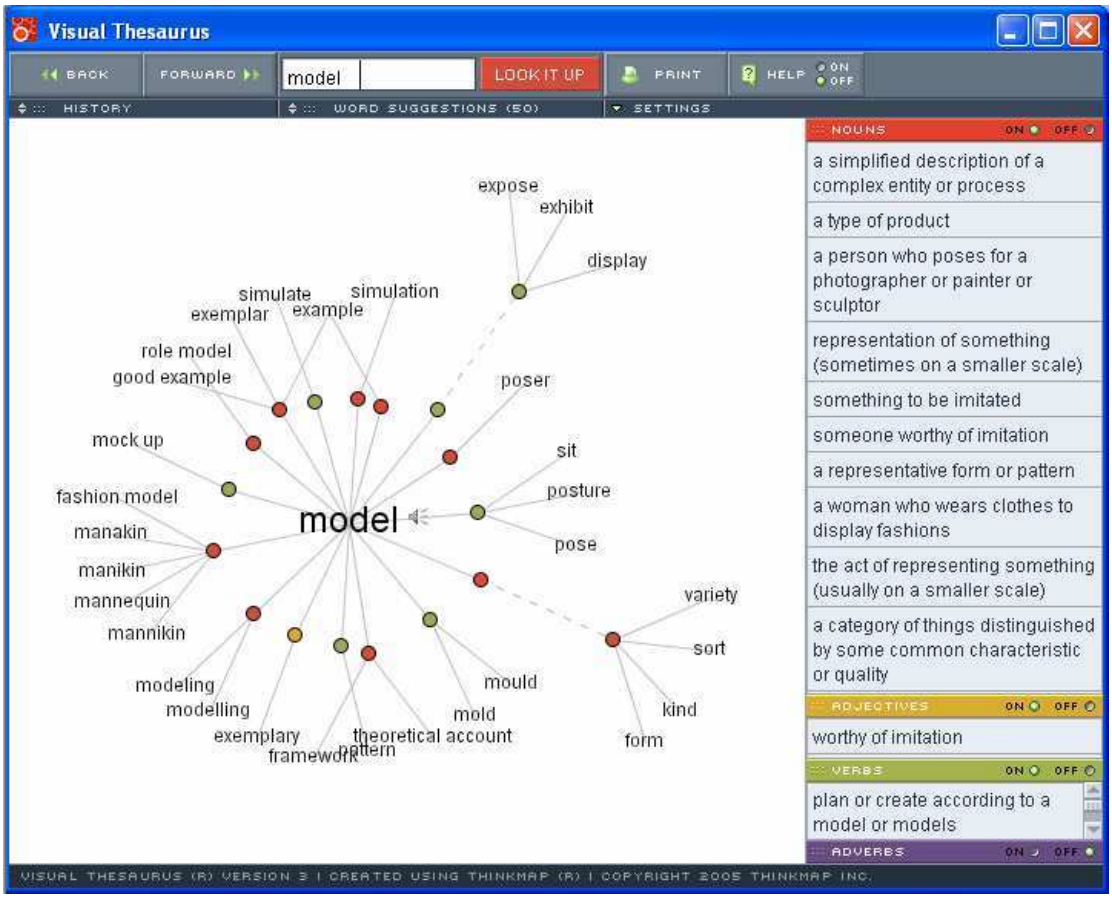
- Vous êtes vous une fois posé la question de l'origine du mode de représentation étrange des personnages dans les fresques égyptiennes? Est-ce un pur hasard ou un choix de modélisation conscient et raisonné ?
- Dans quels langages sont exprimés les modèles suivants ?
- Comment regrouper les différents synonymes de modèles représentés dans le modèle du mot modèle figurant au bas de cette page.





string sexe = «macho» ;





Expériences pour décrire la sémantique en ingénierie des modèles

Benoît Combemale[**] — Sylvain Rougemaille[*] — Xavier Crégut[**]
Frédéric Migeon[*] — Marc Pantel[**] — Christine Maurel[*]

[*] *FéRIA - IRIT-LYRE, Université Paul Sabatier*
118, route de Narbonne
F-31062 Toulouse Cedex 9
{sylvain.rougemaille, frederic.migeon, christine.maurel}@irit.fr

[**] *FéRIA - IRIT-LYRE, INPT-ENSEEIHT*
2, rue Charles Camichel, BP 7122
F-31071 Toulouse Cedex 7
{benoit.combemale, xavier.cregut, marc.pantel}@enseeiht.fr

RÉSUMÉ. *L'Ingénierie Dirigée par les Modèles a permis plusieurs améliorations significatives dans le développement de systèmes complexes en se concentrant sur une préoccupation plus abstraite que la programmation classique. Cependant, il est encore nécessaire d'associer aux éléments de modélisation une sémantique forte pour atteindre un niveau élevé de certification tel que l'exigent actuellement les systèmes critiques. Dans ce cadre, cet article décrit les moyens d'exprimer une sémantique, plus particulièrement opérationnelle, selon deux approches : l'utilisation d'un langage de méta-programmation, Kermeta, et l'utilisation d'un langage de transformation, ATL. Pour ces expérimentations, nous nous appuyons sur un métamodèle simplifié d'un Langage de Description de Procédé.*

ABSTRACT. *MDE has provided several significant improvements in the development of complex systems by focusing on more abstract preoccupation than programming. However, more steps are needed on the semantic side in order to reach high-level certification such as the one currently required for critical embedded systems. In this scope, we present mainly the description of an operational semantics at the meta-model level, following two approaches, using a meta-programming language such as Kermeta and a model transformation Language, ATL. We will base our experimentations on a simple Process description Language metamodel.*

MOTS-CLÉS : *Ingénierie des Modèles, sémantique, OCL, ATL, Kermeta*

KEYWORDS: *Model Driven Engineering, semantics, OCL, ATL, Kermeta*

1. Introduction

L'Ingénierie Des Modèles (IDM) a permis d'établir une nouvelle approche, plus abstraite, pour le développement de systèmes complexes. Un système peut être décrit par différents modèles liés les uns aux autres. L'idée phare est d'utiliser autant de modèles différents (ou Langages Dédiés – Domain Specific Languages) que les aspects chronologiques ou technologiques du développement du système le nécessitent. La principale différence avec un programme est qu'un modèle se concentre sur la syntaxe abstraite alors qu'un programme se concentre sur la syntaxe concrète. La syntaxe abstraite se focalise sur les concepts fondamentaux du domaine et exprime ainsi une sémantique à travers le vocabulaire choisi pour nommer les concepts. Pour les systèmes orientés données, le niveau d'abstraction ainsi obtenu a conduit à des améliorations significatives et semble correspondre à un niveau sémantique adéquat. Pour les systèmes plus orientés calculs, les aspects dynamiques des modèles méritent d'être encore approfondis.

Cette contribution porte sur des approches de définition de la sémantique des méta-modèles en regard des travaux de la communauté des langages de programmation. Les expériences présentées se sont déroulées dans le cadre du projet TOPCASED[1] (Farail *et al.*, 2006) dont le but est de fournir un atelier basé sur l'IDM pour le développement de systèmes logiciels et matériels embarqués. Les autorités de certification pour les domaines d'application de TOPCASED (aéronautique, espace, automobile...) imposent des contraintes de qualification fortes pour lesquelles des approches formelles (analyse statique, vérification du modèle, preuve) sont envisagées.

L'outillage de TOPCASED a pour but de simplifier la définition de nouveaux DSL ou langages de modélisation en fournissant des technologies du méta-niveau telles que des générateurs d'éditeurs syntaxiques (textuels et graphiques), des outils de validation statique et d'exécution des modèles. Cet article présente plusieurs approches pour intégrer les considérations sémantiques des langages de programmation dans une démarche IDM. Nous nous concentrons ici sur deux technologies pour créer des modèles exécutables. Il s'agira dans un premier temps d'expérimenter le langage de méta-programmation Kermeta (Muller *et al.*, 2005) puis, dans un second temps, d'établir une sémantique opérationnelle à partir du langage de transformations de modèles ATL (Atlas Transformation Language) (Bézivin *et al.*, 2005). Nous terminerons par un bilan de ces différentes expérimentations.

Pour illustrer notre approche, nous prendrons l'exemple d'un langage simplifié de description de processus, *SimplePDL* (fig. 1), inspiré du standard SPEM (Obj 2005) proposé par l'OMG. Ce langage définit le concept de processus (*Process*) composé d'un ensemble d'activités (*Activity*) représentant les différentes tâches à réaliser durant le développement. Une activité peut dépendre d'une autre (*Precedes*). Une contrainte sur le démarrage ou la fin de la seconde activité est précisée : *start-to-start*, *finish-to-start* et *finish-to-finish* (*PrecedenceKind*).

1. Officiel : `www.topcased.org` et développement : `gforge.enseeiht.fr/projects/topcased-mm`

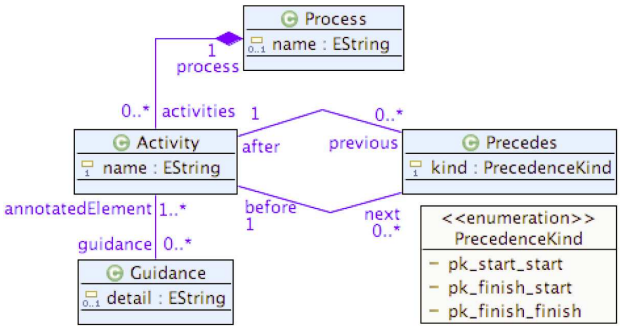


Figure 1. *Le méta-modèle Ecore de SimplePDL*

2. Syntaxe des méta-modèles

2.1. *Définition de la Syntaxe Abstraite*

La syntaxe abstraite d'un langage de modélisation exprime, de manière structurelle, l'ensemble de ses concepts et leurs relations. Les langages de méta-modélisation tels que le standard de l'OMG MOF (Obj 2003a), offrent les concepts et relations élémentaires en termes desquels il est possible de décrire un méta-modèle représentant cette syntaxe abstraite. Nous disposons, à ce jour pour définir cette syntaxe, de nombreux environnement et langages de méta-modélisation : Eclipse-EMF/Ecore (Budinsky *et al.*, 2003), GME/MetaGME(Ledeczi *et al.*, 2001), AMMA/KM3 (ATLAS, 2005) ou XMF-Mosaic/Xcore (Clark *et al.*, 2004).

Pour définir la syntaxe abstraite de *SimplePDL*, nous avons utilisé l'éditeur graphique Ecore du projet TOPCASED permettant la description de méta-modèles à l'aide du langage de méta-modélisation Ecore d'EMF. Nous avons défini le méta-modèle de *SimplePDL* comme le montre la figure 1. Ce méta-modèle représente le point de départ de nos expérimentations. Il sera utilisé comme support pour les diverses expériences que nous présentons ici. En premier lieu, nous désirons donner une syntaxe concrète à *SimplePDL*, afin de pouvoir en créer des modèles conformes.

Cette forme est semblable à la sémantique de construction des arbres abstraits à partir des arbres de syntaxe concrète pour les langages de programmation.

2.2. *Définition de la Syntaxe Concrète*

La syntaxe concrète d'un langage fournit un formalisme, graphique ou textuel, pour manipuler les concepts de la syntaxe abstraite et ainsi en créer des « instances ». Le modèle ainsi obtenu sera conforme au méta-modèle de la syntaxe abstraite. La définition d'une syntaxe concrète ad'hoc est à ce jour bien maîtrisée et outillée. Il

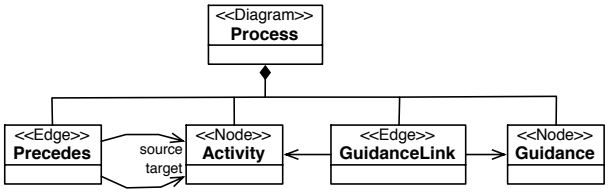


Figure 2. *Modèle de configuration*

existe en effet de nombreux projets qui s'y consacrent, principalement basés sur EMF (Eclipse Modeling Framework) : GMF[2], Merlin Generator[3], GEMS[4], TIGER (Ehrig *et al.*, 2005), etc. Cependant, le défi actuel réside dans l'automatisation de cette concrétisation de la syntaxe. L'idée est de générer automatiquement une syntaxe concrète à partir d'une syntaxe abstraite donnée (Ledeczi *et al.*, 2001, Clark *et al.*, 2004). Cette approche générative, en plus de ses qualités de généricité, permettrait de normaliser la construction des syntaxes concrètes.

TOPCASED propose un outil appelé le « générateur d'éditeur graphique » qui permet, pour un modèle Ecore donné, de définir une syntaxe concrète graphique et l'éditeur associé. Cette génération d'éditeur, s'appuie sur le résultat de la génération de syntaxe textuelle (XML) fournie par EMF (Budinsky *et al.*, 2003) et consiste à définir une correspondance entre les éléments du formalisme graphique (syntaxe concrète) et le modèle Ecore (syntaxe abstraite). Tout ceci est décrit dans le modèle de *configuration* (*configurator*) qui offre une grande liberté de personnalisation des éléments graphiques de la syntaxe concrète souhaitée. Dans TOPCASED, il a été utilisé pour engendrer les éditeurs pour Ecore, UML2, AADL[5] et SAM (formalisme à base d'automates hiérarchiques utilisé par Airbus).

Pour *SimplePDL*, nous avons défini en premier lieu le modèle de notre syntaxe concrète (la figure 2 en présente une version simplifiée). *Activity* et *Guidance* sont définies comme *Node* (boîtes). *Precedes* est définie comme *Edge*, une relation entre deux boîtes. *Process* est représentée comme *Diagram* qui correspond à un paquetage qui contiendra les autres éléments. Définir la syntaxe concrète d'un langage peut nécessiter l'emploi d'éléments additionnels ne correspondant à aucun concept abstrait. Par exemple, ici il est indispensable d'ajouter *GuidanceLink* comme *Edge* pour relier une *Guidance* à une *Activity*. *GuidanceLink* ne correspond pourtant à aucun concept de *SimplePDL*. Sa présence est cependant obligatoire pour lier un élément graphique *Guidance* à la boîte représentant l'activité qu'il décrit. Il se traduit par l'*EReference* appelée *guidance* de *Activity* (fig. 1). Il faut noter que les concepts de la syntaxe abstraite (fig. 1) et ceux de la syntaxe concrète (fig. 2) sont des concepts différents qui

2. Generic Modeling Framework, `http://www.eclipse.org/gmf/tutorial/`
3. `http://sourceforge.net/projects/merlingenerator/`
4. Generic Eclipse Modeling System, `http://sourceforge.net/projects/gems/`
5. Architecture Analysis and Design Language, `http://www.aadl.info/`

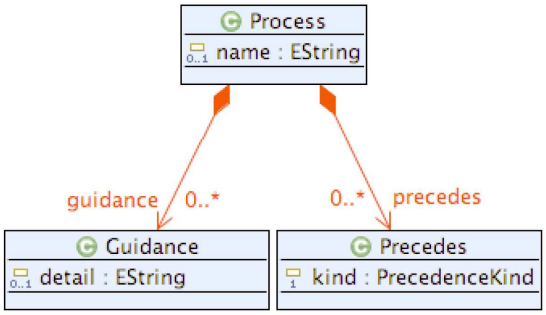


Figure 3. *Extension du méta-modèle pour la description de la syntaxe concrète*

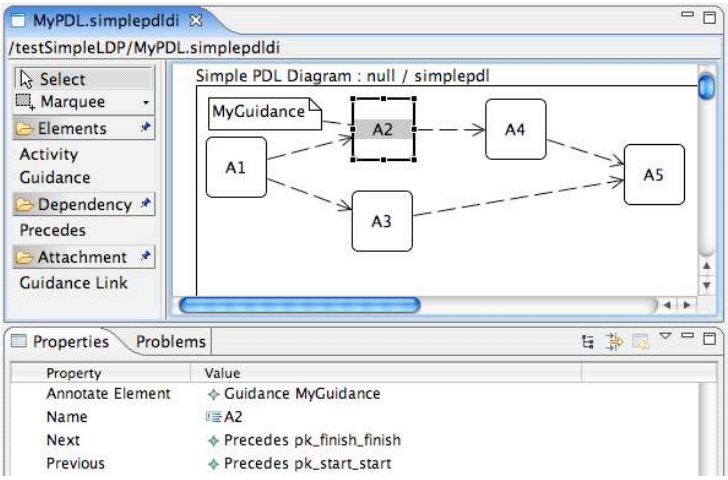


Figure 4. *Éditeur généré, syntaxe concrète graphique de SimplePDL*

doivent être mis en correspondance. Nous avons employé les mêmes noms quand la correspondance était évidente.

Pour pouvoir utiliser le générateur d'éditeur TOPCASED, nous avons dû ajouter deux références (composition), entre *Process* et *Guidance* et entre *Process* et *Precedes* (fig. 3), à notre syntaxe abstraite initiale (fig. 1). Elles sont nécessaires pour pouvoir placer graphiquement une *Activity* et une *Guidance* dans un *Processus*.

La figure 4 présente l'éditeur engendré. Tous les concepts du modèle de configuration sont présents dans la palette. Sélectionner un élément de la palette et le déposer sur le diagramme crée un élément graphique (*node* ou *edge*) et instancie, selon le modèle de configuration, la méta-classe correspondante du méta-modèle *SimplePDL*. Dans le cas d'un « trait », le modèle de configuration permet de préciser sa représenta-

tion graphique, les deux éléments à connecter (source et cible) et l'effet sur le modèle. Par exemple *GuidanceLink* connecte une *Guidance* (source) à une *Activity* (cible) avec un trait interrompu. La *Guidance* est ajoutée dans la EReference *guidances* de l'activité cible. Pour une précédence, les EReferences *next* de l'activité source, *previous* de l'activité cible et *before* et *next* de la *Precedence* sont mises à jour.

3. Sémantique des méta-modèles

L'essentiel des efforts concernant les DSL (Domain Specific Language) s'est porté sur leur aspect syntaxique. Il est donc naturel de se poser la question de la sémantique de tels langages. Par analogie à l'approche suivie pour les langages de programmation, le défi actuel dans l'IDM consiste à donner les moyens d'exprimer une sémantique précise pour les concepts des langages de modélisation. L'ajout de cette sémantique doit être mis en œuvre par des moyens annexes permettant d'enrichir la syntaxe abstraite. On peut classer la sémantique selon les catégories définies pour les langages de programmation (Winskel, 1993) (sémantique axiomatique, dénotationnelle et opérationnelle), et appliquer celle-ci selon les besoins de l'application développée (preuve de modèles, exécution de modèles...) et les outils disponibles.

3.1. *Sémantique axiomatique : preuve de propriétés*

La sémantique axiomatique est basée sur des logiques mathématiques et exprime une méthode de preuve pour certaines propriétés des constructions d'un langages de programmation (Cousot, 1990). Celle-ci peut être très générale (e.g. triplet de Hoare) ou restreinte à la garantie de la cohérence de ces constructions (e.g. typage). Dans le cadre d'un langage de modélisation, cette seconde utilisation est exprimée par le biais de règles de bonne formation, WFR (Well-Formed Rules), au niveau du méta-modèle (fig. 5). Ces règles devront être respectées par les modèles conformes à ce méta-modèle. On peut vérifier le respect de ces règles par analyse statique des modèles. D'autre part, (Steel *et al.*, 2005) propose la notion de type de modèles comme étant l'ensemble des types des objets qu'il contient. Un algorithme de vérification de la compatibilité de deux types est proposé afin de pouvoir, par exemple, s'assurer qu'une transformation peut bien s'appliquer à un modèle donné (type compatible avec le type de modèle source).

Pour exprimer ces règles, l'OMG préconise d'utiliser OCL (Obj 2003b). Appliqué au niveau du méta-modèle, il permet d'ajouter des propriétés, principalement structurelles, qui n'ont pas pu être capturées par la définition du méta-modèle. Il s'agit donc d'un moyen de préciser la sémantique du méta-modèle en limitant les modèles conformes. (Chen *et al.*, 2005) propose dans cet objectif la notion de « set-valued semantics », c'est à dire la restriction de l'ensemble des modèles conformes à un méta-modèle donné, aux seuls modèles respectant les contraintes énoncés au niveau méta.). Par exemple, pour *SimplePDL*, on peut utiliser la contrainte OCL suivante pour imposer l'unicité du nom des activités dans un processus.

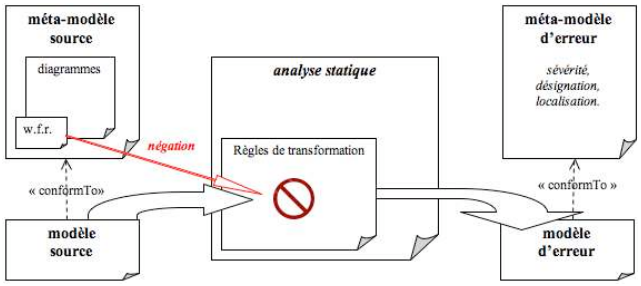


Figure 5. *Vérification de modèles à l'aide d'ATL*

```
context Process inv :
  self.activities->forAll(a1, a2 : Activity |
     a1 <> a2 implies a1.name <> a2.name)
```

Pour vérifier qu'un modèle respecte ces contraintes, on peut utiliser des vérificateurs OCL tels que Use (Richters *et al.*, 2000), OSLO[6], TOPCASED...

Une extension de cette technique, proposée par l'équipe ATLAS du laboratoire LINA (Bézivin *et al.*, 2005), vise à utiliser le langage de transformation de modèles ATL (Jouault *et al.*, 2005) pour engendrer un modèle de diagnostic quand le modèle source contient des erreurs. Ce modèle de diagnostic permet d'obtenir des informations sur l'erreur plus détaillées et plus personnalisées (sévérité, localisation, description...) qu'un simple booléen. Il permet également de mettre en place un système de trace. Le principe est de définir des règles de transformation ATL déclenchées par la reconnaissance de motifs correspondant à la négation d'une WFR (fig. 5). L'exécution de ces règles devra avoir lieu pendant la phase d'analyse statique. Par exemple, la contrainte OCL précédente donnera la règle ci-dessous.

```
rule NomsDifferents {
  from
    p : PDL!Process (
      not ( self.activities->forAll(a1, a2 : Activity |
              a1 <> a2 implies a1.name <> a2.name) )
    )
  to
    pb : Problem!Problem (
      localisation <- p.location,
      sevevrite <- Severity::error,
      description <- 'Deux activités du procédé '
               + p.name + ' ont le même nom.'
    )
}
```

L'exécution d'une telle règle engendre un modèle de diagnostic devant se conformer à un méta-modèle de diagnostic statique (complémentaire à la syntaxe abstraite). L'intérêt de cette approche de vérification est de pouvoir personnaliser le modèle de

6. Open Source Library for OCL http://oslo-project.berlios.de.

diagnostic et ainsi de pouvoir offrir plus de détails et de personnalisation dans le message d'erreur associé à une non-conformité du modèle.

3.2. *Sémantique dénotationnelle : exécutabilité dans un autre formalisme*

Le principe de la sémantique dénotationnelle est de s'appuyer sur un formalisme rigoureusement (i.e. mathématiquement) défini pour exprimer la sémantique d'un langage donné (Mosses, 1990) (Gunter *et al.*, 1990). Une traduction est alors réalisée des concept du langage d'origine vers ce formalisme. C'est cette traduction qui donne la sémantique du langage d'origine.

Dans le cadre de l'IDM, il s'agit d'exprimer des transformations vers un autre espace technique (Blay-Fornarino, 2006), c'est à dire définir un pont entre l'espace technique source et le cible. On parle aussi de sémantique de traduction (Clark *et al.*, 2004). Ces ponts technologiques permettent ainsi d'utiliser les outils de simulation, de vérification et d'exécution fournit par l'espace technique ciblé.

Notons que le travail réalisé pour engendrer avec TOPCASED un éditeur graphique pour *SimplePDL* peut s'apparenter à une dénotation. En effet, partant d'une syntaxe concrète (celle du modèle représenté graphiquement dans l'éditeur), nous avons réalisé une correspondance avec la syntaxe abstraite. Nous avons donc associé aux éléments syntaxiques un concept dans la syntaxe abstraite. Ceci correspond à un début de sémantique.

3.3. *Sémantique opérationnelle : exécutabilité dans le même formalisme*

La sémantique opérationnelle permet de décrire le comportement dynamique des constructions d'un langage. Dans le cadre de l'IDM, elle vise à exprimer la sémantique comportementale d'un méta-modèle afin de permettre l'exécution des modèles qui lui sont conformes.

Pour *SimplePDL*, l'exécution d'un processus consiste à réaliser toutes les activités qui le composent. Le processus est donc terminé quand toutes ses activités sont terminées. Une activité ne peut être commencée que si les activités dont elle dépend sont commencées (précédence *start-to-start*) ou terminées (précédence *finish-to-start*). Au fur et à mesure du développement, le taux de réalisation d'une activité augmente jusqu'à ce qu'elle soit terminée. Elle ne peut être terminée que si les activités précédentes de type *finish-to-finish* sont terminées. Ce sont les équipes de développement qui décident de quelle activité commencer, continuer ou terminer.

Comme le montre la description ci-dessus, exprimer la sémantique opérationnelle nécessite de compléter notre méta-modèle initial (fig. 1) avec les informations liées à l'état des activités. Nous avons donc ajouté l'attribut *progress* sur la méta-classe *Activity* (fig. 6). Elle représente le taux de progression d'une activité : non commencée (-1), en cours (0..99), ou terminée (100).

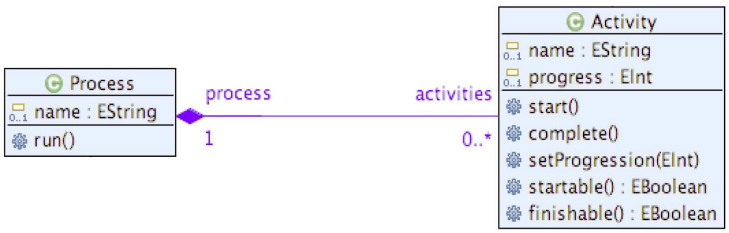


Figure 6. *Méta-modèle étendu pour la définition de la sémantique opérationnelle*

Pour exprimer la sémantique opérationnelle, nous avons envisagé deux approches. La première consiste à définir le comportement de chaque concept de manière impérative à l'aide d'un langage de méta-programmation (Kermeta (Muller *et al.*, 2005), xOCL (Clark *et al.*, 2004)...) ou d'un langage d'action (AS-MOF (Breton, 2002)...). Notre expérimentation a été réalisée avec Kermeta. La seconde approche s'inspire des règles de réduction (Structural Operational Semantics (Plotkin, 1981), sémantique naturelle (Kahn, 1987)) des langages de programmation. Elle consiste à définir le comportement par des transformations endogènes (méta-modèles source et cible identiques) sur le méta-modèle. Celles-ci expriment les modifications de l'environnement d'exécution et du système en fonction de l'état courant du modèle. Nous avons utilisé ATL (Jouault *et al.*, 2005).

3.3.1. *Mise en œuvre avec Kermeta*

Kermeta est défini comme un langage de méta-modélisation exécutable ou de méta-programmation objet : il permet de décrire des méta-modèles dont les modèles sont exécutables. Kermeta est basé sur le langage de méta-modélisation Ecore. Il a été conçu comme un tissage entre un méta-modèle comportemental et Ecore (Muller *et al.*, 2005). En effet, le méta-modèle de Kermeta est composé de deux packages, *Core* et *Behavior* correspondant respectivement à Ecore et à une hiérarchie de métaclasses représentant des expressions impératives. Ces expressions seront utilisées pour programmer la sémantique comportementale des méta-modèles. En effet, chaque concept d'un méta-modèle Kermeta peut contenir des opérations, le corps de ces opérations étant composé de ces expressions impératives (package *Behavior*). On peut ainsi définir la structure aussi bien que la sémantique opérationnelle d'un méta-modèle, rendant ainsi, exécutables les modèles qui s'y conforment.

Kermeta est intégré à l'IDE Eclipse sous la forme d'un plugin et, grâce aux outils de génération Ecore2Kermeta, permet de traduire notre méta-modèle simplifié (fig. 1) en un squelette de code Kermeta. Ce squelette servira de base pour le codage du comportement des modèles *SimplePDL*.

Pour décrire notre sémantique, nous avons ajouté à la syntaxe abstraite, en plus de l'attribut *progress*, des opérations (fig. 6) dont l'exécution fera évoluer notre modèle de procédé. L'opération *run* de *Process* simule le démarrage du procédé en positionnant

à -1 l'attribut *progress* de toutes les activités et gère l'évolution du processus (voir ci-dessous). Les opérations *start*, *setProgression* et *complete* permettent respectivement de démarrer une activité, de changer son taux de progression et de la marquer comme terminée. Ces opérations ne peuvent pas être utilisées à n'importe quel moment. Par exemple, une activité ne peut être commencée que si ses contraintes de précédence sont respectées : les activités précédentes en *start-to-start* sont commencées et celles en *finish-to-start* sont terminées. Nous avons donc ajouté une opération *startable* qui indique si une activité peut être démarrée. De manière analogue, *finishable* indique si une activité peut être terminée (les activités précédentes par *finish-to-finish* sont terminées). On obtient alors le code Kermeta suivant pour quelques unes de ces opérations.

```
operation startable() : Boolean is do
   var start_ok : kermeta::standard::Boolean
   var previousActivities : seq Activity [0..*]
   var prevPrecedes : seq Precedes [0..*]

   if progress==-1 then
      // Getting the activities which have to be started
      prevPrecedes := previous.select{p | p.kind ==
                           PrecedenceKind.pk_start_start }
      previousActivities := prevPrecedes.collect{p | p.before}
      start_ok := previousActivities.forAll{a | a.progress >= 0}
      // Getting the activities which have to be finished
      prevPrecedes := previous.select{p | p.kind ==
                          PrecedenceKind.pk_finish_start }
      previousActivities := prevPrecedes.collect{p | p.before}
      start_ok := start_ok and
               (previousActivities.forAll{a | a.progress==100})
      result := start_ok or (previous.size() == 0)
   else
      result := false
   end
end

operation finishable() : Boolean is do
   var finish_ok : kermeta::standard::Boolean
   var previousActivities : seq Activity [0..*]
   var prevPrecedes : seq Precedes [0..*]
   // Activities must be started
   if progress < 100 and progress >= 0 then
      // Testing previous activities
      prevPrecedes :=
         previous.select{p | p.kind == PrecedenceKind.pk_finish_finish }
      previousActivities := prevPrecedes.collect{p | p.before}
      finish_ok := previousActivities.forAll{a | a.progress==100}
      result := (finish_ok or previous.size()==0)
   else
      result := false
   end
end
```

```
operation start() : Void is do
   if startable() then
      progress := 0
   endif
end
```

```
operation complete() : Void is do
   if finishable() then
      progress := 100
   endif
end
```

Puisque Kermeta permet de programmer des interactions avec l'utilisateur, une interface sommaire a été réalisée. Elle est codée dans la méthode *run* de *Process*.

Elle permet à la personne en charge de gérer le processus de *démarrer une activité* sélectionnée parmi les activités éligibles (*startable*), *changer le taux de progression* d'une activité commencée, *terminer une activité* sélectionnée parmi les activités qui peuvent être terminées (*finishable*) ou *stopper* l'exécution.

3.3.2. *Mise en œuvre avec ATL*

L'intérêt principal de l'utilisation des transformations de modèles endogènes pour l'expression de la sémantique opérationnelle, est de séparer, pour une grande part, la syntaxe abstraite (méta-modèle) de la définition de la sémantique elle même. Ici il sera utilisé comme « donnée » des transformations, toute la sémantique sera contenue dans les transformations et non dans le méta-modèle.

ATL est un langage de transformation de modèles « semi-déclaratif » : il permet de définir une transformation par le biais de règles déclaratives (*rule*) pouvant faire appel à des fonctions auxiliaires (*helper*) qui peuvent être récursives.

Contrairement à Kermeta, il n'est pas possible (et pas souhaitable) de faire d'interaction avec l'utilisateur avec ATL. Aussi, nous avons adapté le modèle d'exécution exploratoire : Une activité est démarrée et terminée dès que possible et le taux de progression d'une activité commencée est augmenté d'une valeur arbitraire (dans notre cas 33) à chaque évolution. L'évolution d'un modèle est décrit par une succession d'appels à une transformation ATL. La seule partie du modèle qui évolue est la valeur de l'attribut *progress* des activités. Aussi cette transformation est constituée de deux règles de copie des éléments *Process* et *Precedes* (*copyProcess* et *copyPrecedes*) et d'une règle d'évolution des activités, *runActivity*, qui recopie tous les attributs de *Activity* et donne sa nouvelle valeur à *progress* en fonction de l'état global du process (taux actuel et état des autres activités).

```
rule runActivity {
  from
    a_in : simplepdl!Activity
  to
    a_out : simplepdl!Activity (
      -- compute new progress value
      progress <-
        if a_in.startable()
        then 0
        else if a_in.progress >= 0
             and a_in.progress < 90
           then a_in.newProgress()
           else if a_in.finishable()
                then 100
                else a_in.progress
                endif
           endif
        endif

      -- copy other attributes
      name <- a_in.name,
      process <- a_in.process,
      previous <- a_in.previous,
      next <- a_in.next,
    )
}
```

Cette règle utilise trois *helper*s : *newProgress* calcule une nouvelle valeur du taux de progression (en fonction de l'évolution arbitraire choisie pour l'expérimentation), *startable* et *finishable* sont équivalentes aux opérations définies en Kermeta. Notons que leur code peut-être écrit de manière plus concise grâce à la possibilité d'utiliser plusieurs fois l'opérateur ->.

```
helper context simplepdl!Activity
   def : startable () : Boolean = (
```

```
      self.progress < 0       -- not already started
      and self.previous->select(p | p.kind = #pk_start_start)
            ->collect(p | p.before)       -- previous start/start
            ->forAll(a | a.progress >= 0)  -- are started
      and self.previous->select(p | p.kind = #pk_finish_start)
            ->collect(p | p.before)       -- previous finish/start
            ->forAll(a | a.progress = 100)  -- are finished
   );

helper context simplepdl!Activity
   def : finishable () : Boolean = (
      self.progress >= 0 and self.progress < 100
      and self.previous->select(p | p.kind = #pk_finish_finish)
            ->collect(p | p.before)       -- previous finish/finish
            ->forAll(a | a.progress = 100)  -- are finished
   );
```

Notons que nous avons choisi de privilégier une exécution particulière pour garder un exemple simple. Pour exprimer la sémantique générale, il faut prévoir de piloter la sémantique par un modèle (les entrées/sorties utilisateurs dans la version Kermeta). Ce point est explicité dans la synthèse suivante.

4. Synthèse

Pour définir une sémantique précise sur un DSL, on peut utiliser plusieurs approches avec à chaque fois plusieurs techniques pour les mettre en œuvre (fig. 7) :

– la *sémantique axiomatique* est exprimée sous forme de règles de bonne formation (WFR) sur le méta-modèle (*MM1, MM2...*), par exemple avec OCL. La vérification statique de ces règles peut être réalisée en utilisant des outils dédiés à ce langage (e.g. USE), ou une transformation vers un modèle de diagnostic (e.g. avec ATL).

– la *sémantique opérationnelle* consiste à rendre exécutables des modèles. Deux approches ont été envisagées. La première consiste à utiliser la méta-programmation avec par exemple Kermeta ou xOCL pour définir un méta-modèle dynamique (*MMx*). De nouvelles opérations sont ajoutées sur le méta-modèle et un langage de comportement permet d'en exprimer le code. La deuxième approche consiste à réaliser une succession de transformations endogènes. Nous les avons exprimées en utilisant ATL. Dans le paragraphe suivant, nous revenons sur ces deux techniques.

– enfin, la *sémantique dénotationnelle* que nous n'avons pas encore expérimentée vise un objectif similaire à la sémantique opérationnelle mais en « traduisant » le modèle d'origine *M* vers un modèle *Msem* conforme à un méta-modèle exécutable *MMsem* (*Uppaal*, *IF*...). Cette traduction *Mt1* est bien sûr faite en s'appuyant sur les méta-modèles correspondants. Une transformation *Mt2* doit permettre de traduire les résultats de l'exécution/simulation de *Msem* en des concepts définis dans *MMx*.

Dans cet article, nous avons surtout détaillé les sémantiques axiomatique et opérationnelle. Les expérimentations réalisées sur un exemple simple ont permis de confirmer les éléments apparus dans le cadre des langages de programmation :

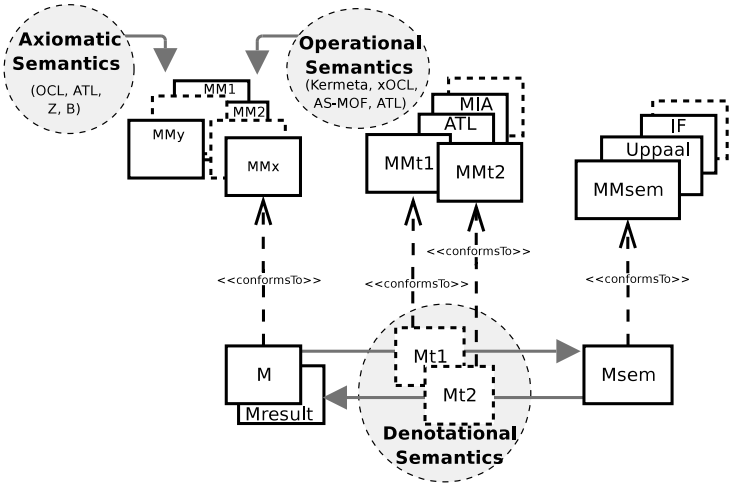


Figure 7. *Approches pour définir la sémantique des méta-modèles*

– la sémantique s'exprime au niveau de la syntaxe abstraite. La syntaxe concrète n'est qu'une forme équivalente plus adaptée à l'exploitation par l'utilisateur. L'avantage des modèles est le grand nombre de syntaxes concrètes (e.g. textuelles pour l'utilisation, graphiques pour la présentation) qui peuvent être associées à une même syntaxe abstraite ;

– La syntaxe abstraite représente les concepts d'un langage de modélisation dont il est envisageable de définir l'exécution (selon la nature du domaine). Elle ne contient donc pas nécessairement les éléments permettant l'expression des étapes de cette exécution, c'est-à-dire la représentation complète de l'état du système à chacune de ces étapes. Le méta-modèle initial, que nous appelons méta-modèle statique, doit donc être compléter par un méta-modèle permettant de capturer ces étapes intermédiaires ; nous l'appelons méta-modèle dynamique. C'est-à-dire un méta-modèle contenant tout les concepts rendant compte des modifications de l'état d'un modèle au cours de son exécution (e.g. notion d'état courant). Pour être puriste, il faudrait considérer que le méta-modèle dynamique contient une copie du méta-modèle statique avec des concepts de nom différent. Notons que le méta-modèle dynamique doit également contenir la représentation des erreurs d'exécution ;

– La sémantique dynamique représente l'état interne du système. Il est également nécessaire de représenter les échanges avec l'environnement extérieur au système. Il faut donc également introduire un méta-modèle que nous appelons d'interaction. Celui-ci devant définir les concepts liés à l'environnement d'exécution (e.g. entrées/-sorties, mémoire etc.), permettant ainsi de représenter plus fidèlement l'exécution du système qu'il représente ;

– Le méta-modèle complet (statique + dynamique + interaction) permet de représenter les différentes étapes dans la vie d'un système. Les transformations de modèles

« à la » ATL ou les opérations sémantiques « à la » Kermeta expriment alors les étapes de cette exécution. Il peut être nécessaire de pouvoir conserver les différentes étapes intermédiaires dans une exécution. Il faudrait alors introduire un méta-modèle que nous appelons de trace visant à conserver la séquence des événements liés à l'exécution d'un modèle et par exemple, autoriser le « re-jeu ». Pour l'instant ce méta-modèle n'a pas encore fait l'objet d'expérimentations.

La définition d'une sémantique opérationnelle au niveau méta-modèle repose donc, d'une part sur l'extension du méta-modèle statique par les méta-modèles dynamique, d'interaction et de trace, et d'autre part par l'expression des mécanismes d'exécution sous la forme de règles de transformation ou d'opérations d'exécution des modèles étendus.

Pour la sémantique opérationnelle, nous avons expérimenté deux approches : l'utilisation d'un langage de méta-modélisation (méta-programmation avec Kermeta) ou la définition de transformations (ATL). Comparons les avantages et les inconvénients de ces deux approches.

– L'utilisation de transformations de modèles endogènes permet de séparer nettement la description du système et ses règles d'évolution. Cette séparation existe également dans l'approche méta-programmation mais elle est moins nette. Cette séparation peut être utile pour simplifier le raisonnement sur les propriétés des modèles et de leurs sémantiques. Elle facilite également la définition de plusieurs sémantiques pour un même méta-modèle. Sachant qu'il suffit dans la méta-programmation de dériver plusieurs méta-modèles avec des opérations différentes.

D'autre part, l'utilisation de transformations déclaratives qui lisent un modèle dans un certain état et produisent un nouveau modèle dans un nouvel état facilite l'utilisation d'un raisonnement par induction ou co-induction sur la structure des modèles. Ceci est nécessaire lorsque des propriétés doivent être prouvées sur la sémantique d'un modèle particulier, ou sur la sémantique elle-même.

– L'utilisation d'opérations permet d'exprimer plus facilement des évolutions complexes en utilisant des instructions sophistiquées. Celle-ci permettent également de réaliser des effets de bord sur le contenu d'un modèle et de ne changer que les parties concernées par une certaine évolution. La définition de la sémantique est donc en général plus simple pour l'utilisateur.

Enfin, l'approche méta-programmation semble plus expressive pour l'utilisateur alors que l'approche « transformation » est plus abstraite et adaptée au raisonnement. Notons également qu'il s'agit d'une question de style. L'approche à base de transformation est plutôt déclarative et l'approche méta-programmation plutôt impérative.

5. Travaux Relatifs

La définition d'une sémantique formelle des langages de modélisation est actuellement un objectif important du monde de l'IDM. Outre Kermeta et ATL qui offrent

des outils pour appréhender ces aspects, nous avons relevé d'autres projets qui traitent de ce problème important.

Le laboratoire ISIS de l'Université de Vanderbilt s'intéresse à l'ingénierie des modèles depuis plusieurs années. On y prône les principes du MIC (Model-Integrated Computing), dans lequel les modèles sont au centre du développement de logiciels intégrés. GME (Ledeczi *et al.*, 2001) y a été développé pour permettre de décrire des modèles de DSL hiérarchiques et multi-aspects. Cet objectif a confronté l'équipe de GME au problème évoqué dans cet article, à savoir la définition d'une sémantique précise pour un DSL. Elle a récemment proposé, en réponse à cette question, un ancrage sémantique dans un modèle formel bien formé, le modèle ASM (Abstract State Machine) (Gurevich, 2001), en utilisant le langage de transformation GReAT (Graph Rewriting And Transformation language) (Agrawal *et al.*, 2005).

Xactium[7] est une société créée en 2003 dont les objectifs sont de fournir des solutions pratiques pour le développement de systèmes imposants selon les principes de l'ingénierie des modèles. Elle a développé l'outil XMF-Mosaic (Clark *et al.*, 2004) qui permet de définir un DSML, de simuler et de valider des modèles au moyen d'une extension du langage OCL appelée xOCL (eXecutable OCL). XMF-Mosaic offre aussi, les moyens de transformer des modèles et de définir des correspondances avec d'autres outils de manipulation de modèles.

Ces travaux sont très proches des objectifs de l'environnement TOPCASED, i.e. proposer un environnement de modélisation modulable basé sur une approche générative (comme GME, XMF), accompagné d'un sémantique formelle offrant des moyens de simulation et de validation de modèles. Nos travaux autour de Kermeta s'inscrivent dans un cadre proche de celui d'xOCL au sein de l'outil XMF-Mosaic. La notion d'ancrage sémantique proposée par le laboratoire ISIS est, quant à elle, assimilable à la sémantique dénotationnelle telle que nous la proposons. La différence est que l'on souhaite autoriser plus de liberté dans le choix du modèle sémantique et valoriser les retours de simulations ou de vérifications au sein d'un modèle particulier (*Mresult*, fig. 7). En revanche, ces travaux ne proposent pas l'utilisation de règles de réécriture de modèles pour la définition de la sémantique opérationnelle.

6. Conclusion et perspectives

Cet article présente les moyens d'exprimer une sémantique forte pour les méta-modèles. Nous détaillons pour cela plusieurs approches dérivées des travaux de la communauté des langages de programmation. Nous nous sommes concentrés sur la définition de méta-modèles exécutables pour un sous-ensemble simplifié d'un langage de description de procédé. Nos travaux s'appuient sur le langage de méta-programmation Kermeta et sur le langage de transformation de modèle ATL.

7. http://www.xactium.com

Ces travaux ouvrent de nouvelles perspectives. Premièrement, en conservant l'approche de transformation de modèles endogènes, il convient d'étudier si les outils des langages de programmation tels que la séparation des préoccupations et le tissage permettraient d'exprimer de manière séparée les différents modèles de la figure 7. On pourrait alors définir la transformation comme un tissage entre plusieurs types de méta-modèles (statique, dynamique, de diagnostic, d'interaction...). Une deuxième piste sera d'explorer la définition dénotationnelle de la sémantique. Dans ce cas, la difficulté n'est pas la transformation de modèles à proprement parler et le moyen de l'exprimer ; il s'agit surtout de régler le problème du changement de langage et de l'inversibilité de la transformation. Ce travail est à corréler aux travaux que nous démarrons dans le cadre de TOPCASED concernant la simulation de modèles pour lesquels les problématiques sont les mêmes.

7. Bibliographie

Agrawal A., Karsai G., Kalmar Z., Neema S., Shi F., Vizhanyo A., The Design of a Language for Model Transformations, Technical report, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37235, USA., 2005.

ATLAS, KM3 : Kernel MetaMetaModel, Technical report, LINA & INRIA, Nantes, aug, 2005.

Bézivin J., Jouault F., « Using ATL for Checking Models », *GraMoT*, 2005.

Blay-Fornarino J.-M. F. J. E. M., *L'ingénierie dirigée par les modèles. Au-delà du MDA*, Hermes Sciences / Lavoisier, 2006.

Breton E., Contribution à la représentation de processus par des techniques de méta-modélisation, PhD thesis, Université de Nantes, jun, 2002.

Budinsky F., Steinberg D., Ellersick R., *Eclipse Modeling Framework : A Developer's Guide*, Addison-Wesley Professional, 2003.

Chen K., Sztipanovits J., Abdelwalhed S., Jackson E., « Semantic Anchoring with Model Transformations », *in* , S.-V. LNCS 3748 (ed.), *Model Driven Architecture - Foundations and Applications, First European Conference (ECMDA-FA)*, p. 115-129, 2005.

Clark T., Evans A., Sammut P., Willans J., « Applied Metamodelling - A Foundation for Language Driven Development », 2004, version 0.1.

Cousot P., « Methods and Logics for Proving Programs. », *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, p. 841-994, 1990.

Ehrig K., Ermel C., Hänsgen S., Taentzer G., « Towards Graph Transformation Based Generation of Visual Editors Using Eclipse », *Electr. Notes Theor. Comput. Sci*, 2005.

Farail P., Gaufillet P., Canals A., Camus C. L., Sciamma D., Michel P., Crégut X., Pantel M., « the TOPCASED project : a Toolkit in OPen source for Critical Aeronautic SystEms Design », *Embedded Real Time Software (ERTS)*, Toulouse, 25-27 January, 2006.

Gunter C. A., Scott D. S., « Semantic Domains. », *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, p. 633-674, 1990.

Gurevich Y., « The Abstract State Machine Paradigm : What Is in and What Is out », *Ershov Memorial Conference*, 2001.

Jouault F., Kurtev I., « Transforming Models with ATL », *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica, 2005.

Kahn G., Natural Semantics, Report no. n° 601, INRIA, February, 1987.

Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., IV C. T., Nordstrom G., Sprinkle J., Volgyesi P., « The Generic Modeling Environment », *Workshop on Intelligent Signal Processing*, Budapest, Hungary, May 17, 2001.

Mosses P. D., « Denotational Semantics. », *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, p. 575-631, 1990.

Muller P.-A., Fleurey F., Jézéquel J.-M., « Weaving Executability into Object-Oriented Meta-Languages », *LNCS*, MODELS/UML'2005, Springer, Montego Bay, Jamaica, october, 2005.

Obj, *Meta Object Facility (MOF) 2.0 Core Specification*. oct, 2003a.

Obj, *UML Object Constraint Language (OCL) 2.0 Specification*. oct, 2003b, Final Adopted Specification.

Obj, *Software Process Engineering Metamodel (SPEM) 1.1 Specification*. jan, 2005, formal/05-01-06.

Plotkin G. D., A Structural Approach to Operational Semantics, Technical Report n° FN-19, DAIMI, University of Aarhus, Denmark, September, 1981.

Richters M., Gogolla M., « Validating UML Models and OCL Constraints », *in* , A. Evans, , S. Kent, , B. Selic (eds), *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference*, vol. 1939 of *LNCS*, Springer Verlag, p. 265-277, octobre, 2000.

Steel J., Jézéquel J.-M., « Model Typing for Improving Reuse in Model-Driven Engineering », *in* , L. C. Briand, , C. Williams (eds), *MoDELS*, vol. 3713 of *Lecture Notes in Computer Science*, Springer, p. 84-96, 2005.

Winskel G., *The formal semantics of programming languages : an introduction*, MIT Press, Cambridge, MA, USA, 1993.

ANNEXE POUR LE SERVICE FABRICATION

A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LES ACTES :
   IDM'06, Lille – Secondes Journées sur l'ingénierie des modèles

2. AUTEURS :
   Benoît Combemale[**] — Sylvain Rougemaille[*] — Xavier Crégut[**]
   Frédéric Migeon[*] — Marc Pantel[**] — Christine Maurel[*]

3. TITRE DE L'ARTICLE :
   Expériences pour décrire la sémantique en ingénierie des modèles

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   Sémantique et IDM : expériences

5. DATE DE CETTE VERSION :
   7 juin 2006

6. COORDONNÉES DES AUTEURS :

   – adresse postale :

   [*] FéRIA - IRIT-LYRE, Université Paul Sabatier
   118, route de Narbonne
   F-31062 Toulouse Cedex 9
   {sylvain.rougemaille, frederic.migeon, christine.maurel}@irit.fr

   [**] FéRIA - IRIT-LYRE, INPT-ENSEEIHT
   2, rue Charles Camichel, BP 7122
   F-31071 Toulouse Cedex 7
   {benoit.combemale, xavier.cregut, marc.pantel}@enseeiht.fr

   – téléphone : 05 61 58 80 37

   – télécopie : 05 61 58 83 06

   – e-mail : benoit.combemale@enseeiht.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes.cls`,
   version 1.2 du 2004/09/07.

8. FORMULAIRE DE COPYRIGHT :
   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com

Actes de l'atelier de travail
Motifs de méta-modélisation

Un méta-métamodèle pour la gestion de modèles

Thi-Lan-Anh Dinh*, — Olivier Gerbé** — Houari Sahraoui*,*****

** Département d'informatique et de recherche opérationnelle*
Université de Montréal
CP 6128 succ. Centre Ville, Montréal, Québec, Canada, H3C 3J7
{dinhthil, sahraouh}@iro.umontreal.ca

***HEC Montréal*
3000 Chemin de la Côte-Sainte-Catherine, Montréal, Québec, Canada H3T 2A7
{lan-anh.dinh-thi, olivier.gerbe}@hec.ca

****Escuela Superior de Informática, UCLM*
Paseo de la Universidad, 4.13071 Ciudad Real, España

RÉSUMÉ. *Les nombreux travaux de recherche autour de IDM (Ingénierie Dirigée par les Modèles) montrent que la gestion de modèles intéresse de nombreuses communautés de recherche. En effet, dans les domaines de la gestion des connaissances, la gestion de méta-données, les ontologies, la qualité de service et le génie logiciel, les chercheurs travaillent beaucoup sur la modélisation et la métamodélisation. La représentation de modèles et de métamodèles est donc un axe de recherche essentiel pour la gestion de modèles. Dans ce contexte, nous proposons ici un méta-métamodèle réflexif qui permet de décrire divers modèles et métamodèles.*

ABSTRACT. *Research work in the fields of Model-Driven Engineering (MDE) and Model-Driven Development (MDD) shows that model management attracts special interest from different research communities. The communities of knowledge management, metadata management (databases), ontologies, quality of service, software engineering, etc. are particularly focusing on the definition and the use of models. Therefore, representing models is an essential research axis for model management. In this context, we propose a reflexive meta-metamodel allowing to describe diverse metamodels.*

MOTS-CLÉS : *Modélisation, Méta-modélisation, Modèle, Métamodèle, Méta-métamodèle.*

KEYWORDS: *Modeling, Meta-modeling, Model, Metamodel, Meta-metamodel.*

1. Introduction

La notion de modèles fait l'objet de discussions depuis l'antiquité. Cependant, dans la communauté informatique, l'émergence d'une vision de développement du logiciel appelé *Model Driven Engineering* - Ingénierie Dirigée par les Modèles ou IDM - a relancé l'intérêt en mettant l'accent sur les activités de modélisation et de gestion de modèles. La gestion de modèles traite des mécanismes qui permettent de représenter, créer, stocker et manipuler les modèles. Elle intervient dans des domaines aussi divers que le génie logiciel, les bases de données, la qualité de service et la gestion des connaissances.

En génie logiciel, l'approche Ingénierie Dirigée par les Modèles (Bézivin *et al.*, 2005; Girard *et al.*, 2005; Jézéquel *et al.*, 2005) qui a succédé au *Model Driven Architecture* (MDA) (OMG, 2003a) a pour objectif de définir un cadre pour la génération de code par des transformations successives de modèles.

Dans le domaine des bases de données, la gestion des méta-données touche la manipulation de la structure des données plutôt que les données elles-mêmes. L'approche basée sur la gestion de modèles est une solution possible aux problèmes d'intégration et transformation de données (Alagic *et al.*, 2001; Bernstein *et al.*, 2000; Melnik, 2004).

Dans le domaine de la qualité de service, afin d'assurer le fonctionnement des applications dans un environnement complexe tel que des systèmes multimédias répartis, l'intégration de l'information de gestion devient fondamental (Kerhervé *et al.*, 2001). Une solution possible est de développer des modèles génériques pour les systèmes, les applications et les utilisateurs et d'ensuite développer des mécanismes d'intégration (Gerbé *et al.*, 2003a).

Dans le domaine de la gestion des connaissances, l'intérêt sémantique de la modélisation prime sur l'intérêt d'opérationnalisation. Ce qui est recherché avant tout dans ce domaine, c'est la compréhension du monde à modéliser et une représentation ou documentation la plus précise mais aussi la plus souple possible. La gestion des connaissances dans les entreprises, c'est-à-dire le développement de mémoires corporatives pose principalement un problème de quantité, de complexité et de diversité (Dinh *et al.*, 2004). Ceci implique un véritable défi pour la représentation et la modélisation de ces mémoires corporatives.

Dans sa thèse (Schneider, 1994), Daniel K. Schneider discute la notion de modèle scientifique avec une approche sciences sociales qui apporte un éclairage intéressant à ceux qui s'intéresse comme nous à la modélisation dans une perspective gestion de connaissances. Schneider reprend à son compte les trois fonctions d'un modèle développé par Stachowiak (Stachowiak, 1965). Un modèle a une fonction de représentation; un modèle représente un original naturel ou artificiel que l'on peut décrire comme un ensemble d'éléments et leurs interrelations. Un modèle a une fonction de réduction; un modèle ne représente que les caractéristiques pertinentes au but de la modélisation. Enfin, un modèle a une

fonction « subjectivisante »; un modèle n'a pas de relation « naturelle » avec son original, l'interprétation d'un modèle tient compte du but et de l'usage.

La représentation de modèles est un axe de recherche essentiel dans la gestion des modèles de connaissances. Selon nos études (Dinh *et al.*, 2005), les formalismes couramment utilisés en modélisation (graphes conceptuels, sNets, CDIF, MOF, OWL) ne permettent pas de satisfaire tous les besoins essentiels pour la représentation et la gestion de modèles dans le cadre de la gestion de connaissances. Le lecteur trouvera à la fin de l'article, après une mise en contexte (section 3 et section 5), le résumé de ces études.

De plus, il est à noter que l'utilisation d'un ensemble de modèles avec différents formalismes pose souvent des problèmes de traduction et de pertes d'information lors des échanges entre modèles bien que ces modèles soient compatibles. Notre objectif est donc de spécifier un nouveau formalisme qui sera susceptible de satisfaire tous les besoins de la gestion de modèles dans le cadre de la gestion de connaissances. Ce formalisme est principalement basé sur les réseaux sémantiques (Sowa *et al.*, 1991) en souhaitant de maintenir la flexibilité des réseaux sémantiques dans l'expression ainsi que la simplicité dans la représentation pour la facilité de mise en œuvre.

Cet article présente une première ébauche de ce formalisme et traite principalement de la représentation des métamodèles. Les métamodèles sont des modèles particuliers permettant de décrire les modèles eux-mêmes. Le reste de l'article est organisé comme suit. La section 2 décrit l'architecture de modélisation et métamodélisation et les différents niveaux de modélisation. Dans la section 3, les notions de base et les besoins essentiels pour un méta-métamodèle sont présentés. Nous présentons notre méta-métamodèle dans la section 4 et l'évaluons en le comparant avec d'autres formalismes dans la section 5. Finalement, la section 6 conclut et présente nos travaux futurs.

2. Architecture de modélisation

L'architecture de modélisation est basée sur quatre niveaux. Cette architecture est largement acceptée aujourd'hui (Sahraoui, 1995; Revault *et al*., 1995; Lemesle, 2000; Bézivin *et al.*, 2001; Dinh, 2003; Girard *et al.*, 2005). Nous présentons brièvement ci-dessous chacun des quatre niveaux :

– M3 (méta-métamodèle) est le niveau le plus abstrait et réflexif dans cette architecture. Il définit les notions de base permettant la représentation des niveaux inférieurs ainsi que lui-même.

– M2 est le niveau métamodèle. Ce niveau définit le langage et la grammaire pour représenter des modèles au niveau M1.

– M1 est le niveau modèle. Ce niveau définit des représentations concrètes du monde réel (modèles) ainsi que des descriptions de ces représentations. Chaque

modèle au niveau M1 respecte la grammaire spécifiée par son métamodèle au niveau M2.

– M0 est le monde réel décrit au niveau M1.

Dans cette architecture, seuls les trois premiers niveaux (M3, M2, M1) appartiennent au monde de la modélisation, le niveau M0 est le monde réel. Ce que l'on appelle couramment les types et les instances sont au même niveau M1 (Lemesle, 2000; Bézivin *et al.*, 2001). Afin d'éviter des problèmes dus à la représentation de la nature des concepts (y compris des modèles), par exemple le problème de l'instanciation double discuté dans (Bézivin *et al.*, 2001; Dinh *et al.*, 2004), une architecture de modélisation devrait bien distinguer la notion d'instanciation, la notion de conformité entre éléments et la notion de conformité entre modèles. Ces notions sont en effet de nature très différente, chacune possède son propre comportement vis-à-vis des contextes (global ou local) dans lesquels elle se situe. L'instanciation indique le rapport «types-instances» entre éléments du niveau M1. La conformité entre éléments indique le rapport de conformité entre un élément et son méta-élément. Enfin, la conformité sémantique entre modèles indique le rapport de conformité sémantique entre un modèle et son métamodèle. La Figure 1 illustre un exemple de ces différentes notions. Dans cette figure, *Marie* au niveau M1, représente la vraie personne *Marie* (au niveau M0). *Marie* au niveau M1 est d'une part conforme à *Object* dans le contexte global (ce qui est spécifié par la relation de type $meta_{M2}$ sur l'axe vertical), et d'autre part une instance de *Personne* dans le contexte local (ce qui est spécifié par la relation de type *instOf* sur l'axe horizontal). La Figure 1 montre aussi sur l'axe vertical par la relation de type *sem* que M3 est conforme sémantiquement à lui-même et que M2 est conforme sémantiquement à M3 et par la relation de type sem_{M2} que M1 est conforme sémantiquement au métamodèle M2. Notons que la réflexivité du niveau le plus haut (M3 dans ce cas) possède des avantages multiples. Elle permet de limiter le nombre de niveaux d'abstraction. Le niveau réflexif se valide lui-même; les outils et algorithmes applicables au niveau métamodèle sont donc aussi applicables à ce niveau (Lemesle, 2000). Ceci facilite l'adaptation aux extensions ou modifications futures.

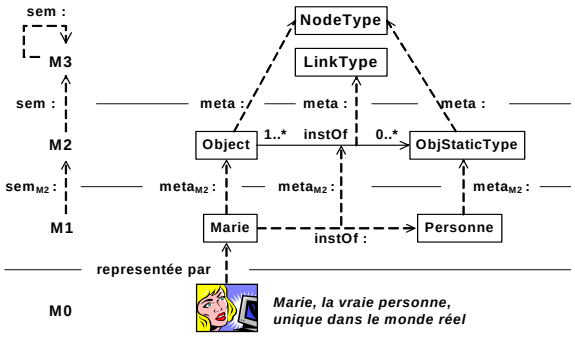


Figure 1. *Notre architecture de modélisation*

3. Besoins pour M3

Cette section recense les besoins essentiels pour un M3. En pratique, les choses à modéliser sont représentées par des éléments et des relations entre eux. Nous considérons ici qu'une relation n-aire peut toujours être décomposée en relations binaires.

Nous détaillons ci-dessous les principaux besoins. Le besoin 1 présente les notions de base. Le besoin 2 traite du passage d'un niveau de modélisation à un autre. Le besoin 3 a pour fin de classer les types non relationnels et les types relationnels. Le besoin 4 représente l'exigence de représentation des contraintes de cardinalités qui fait partie de la représentation sémantique des relations. De plus, pour la représentation et la gestion de modèles, nous avons les besoins 5 et 6. Le besoin 5 exprime la nécessité d'expliciter différents types de modèles. Le besoin 6 exprime le besoin de représentation des interactions possibles entre modèles.

Besoin 1. *Notions de base*. Un *type* représente un ensemble de choses ayant les mêmes propriétés. Une *instance* d'un type représente une chose qui se conforme à la définition du type. Parmi les types, il convient de distinguer les *types relationnels* (associations) et les types non relationnels (classe, entité). Parmi les instances, il convient de distinguer les *liens* – instances des types relationnels – et les occurrences – instances des types non relationnels.

Besoin 2. *« Type » ou « Instance »* (Gerbé, 2000; Dinh *et al.*, 2004). Un élément peut être traité comme « instance » à un niveau mais comme « type » à un niveau inférieur. Par exemple, *ObjStaticType* est vu comme une « instance » de *NodeType* et comme le « type » de *Personne*.

Besoin 3. *Hiérarchisation entre types non relationnels et hiérarchisation entre types relationnels*. Les types relationnels et les types non relationnels sont organisées en hiérarchie.

Besoin 4. *Contraintes de cardinalité maximale/minimale pour les types relationnels*. Ceci capture le nombre des relations possibles par rapport aux éléments impliqués dans une relation.

Besoin 5. *Distinction entre modèles de différentes natures*. Exemples : *métamodèle* d'un modèle; *modèle structurel* d'un type relationnel (qui spécifie comment ce type relationnel relie ses éléments); *modèle de conditions* (pour contextualiser des conditions).

Besoin 6. *Relations avec les modèles :* relations de *contenant* entre un modèle et ses éléments; relations *d'accès* entre modèles pour réutiliser dans un modèle des éléments d'un autre modèle; relation de *respect sémantique* entre un modèle et son métamodèle; et d'autres relations comme *l'intersection, la différence, l'inférence, la restriction*.

4. Méta-métamodèle (M3)

Le méta-métamodèle (M3) fournit le langage et la grammaire pour décrire les formalismes de modélisation. Notre méta-métamodèle est inspiré de celui des graphes conceptuels (Gerbé, 2000; Sowa, 1984) et des sNets (Lemesle, 2000).

Comme dans les réseaux sémantiques (Lemesle, 2000; Sowa *et al.*, 1991), la représentation est basée sur les nœuds et les arcs. Un nœud représente une occurrence alors qu'un arc représente un lien. Un nœud est représenté graphiquement par un rectangle alors qu'un arc est représenté graphiquement par un arc orienté, en pointillé.

4.1. *Éléments de base de M3*

Les éléments de base du niveau M3 sont les types : *KO*, *T*, *R*, *Type*, *NodeType*, *LinkType* et les relations : *sub*, *meta*, *srce* et *dest*. La Figure 2 présente la hiérarchie de généralisation de ces éléments.

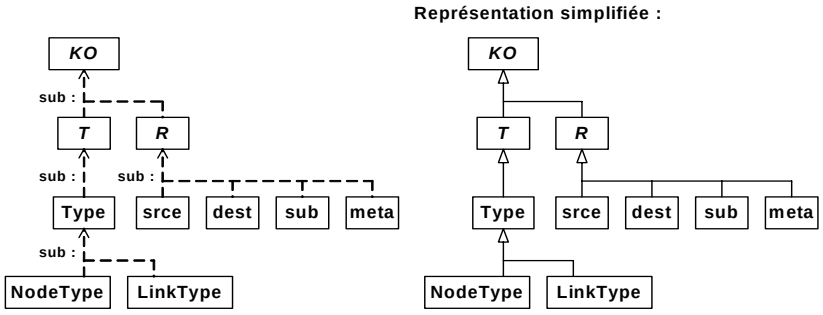


Figure 2. *Hiérarchie des éléments de base de M3*

KO, T, R. *KO* (*Knowledge Object*) est à la racine de la hiérarchie de tous les types définis au niveau M3. Il est l'élément universel et représente l'ensemble des tous les objets. Cet ensemble est subdivisé en deux sous-ensembles: les noeuds (*T*), et les liens (*R*). *KO*, *T*, *R* sont des types abstraits, ils n'ont pas directement d'instances et servent à éclaircir sémantiquement la hiérarchie de spécialisation des types.

Type, NodeType, LinkType. *Type* représente l'ensemble de tous les types qui sont subdivisés en deux sous-ensembles : les *NodeType*, et les *LinkType*. Les types du niveau M3 permettent de définir les éléments (y compris les types) du niveau M2. Un *NodeType* représente un type non relationnel, un type de noeud dit *méta-nœud*. Un *LinkType* représente un type relationnel, un type de realtion binaire et

directionnelle dit *méta-lien* entre deux types. Un *LinkType* est défini par un *Type* source et un *Type* destination (Figure 3).

srce, dest. *srce* et *dest* permettent de spécifier les sources et destinations de *LinkType*. Comme le montre la Figure 3, le méta-lien *srce* (rectangle) lie *Type* et *LinkType*. Le méta-lien *srce* a pour source *Type* et destination *LinkType* à travers un lien de type *srce* et un lien de type *dest* (flèches pointillées). Il s'agit de la définition de tous les méta-liens. Tous les éléments conformes à *LinkType*, c'est-à-dire, *R* et ses sous-types (y compris *srce*, *dest*, *meta*, *sub*) se conforment à cette définition.

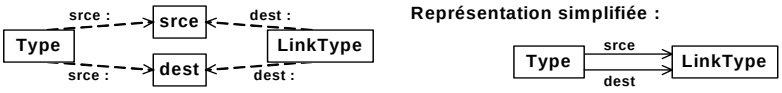


Figure 3. *Structure initiale de srce, dest*

sub. *sub* sert à classifier les types. Il implémente la relation de sous-typage entre types (Besoin 3). L'effet de cette relation (cf. les Règle 1 et Règle 2 ci-dessous) nous permet, en particulier, de déduire toutes les structures nouvelles possibles pour un méta-lien à partir de sa structure initiale en remplaçant dans cette structure l'élément source ou destination par un de ses inférieurs. Comme *NodeType* et *LinkType* sont des inférieurs de *Type*, la structure initiale de *sub* (Figure 4) autorise l'existence des liens de type *sub* entre les *Type*, y compris les *NodeType* et les *LinkType*. De plus, les définitions des types *srce* et *dest* (Figure 3) nous autorisent à définir d'autres structures de *LinkType*. Un *LinkType* peut, à travers un *srce* et un *dest*, unir deux *NodeType* ou un *Type* avec un *NodeType*, ou deux *LinkType*, ou *NodeType* avec un *LinkType*. Ceci implique que nous pouvons définir toutes les sortes de liens : un lien entre deux noeuds, ou entre deux liens, ou entre un noeud et un lien. Le pouvoir d'expression du formalisme augmente alors considérablement. Le méta-lien *sub* est cependant restreint par la Règle 3, présentée un peu plus loin.

Figure 4. *Définition de sub*

meta. Les liens de type *meta* représentent les relations de conformité qui associent des éléments à leurs méta-éléments, soit du même niveau M3 ou du niveau M2 au niveau M3 dans l'architecture de modélisation. Dans le deuxième cas, ces liens jouent le rôle de transition entre les niveaux M2 et M3. Un élément (noeud ou lien) au niveau M3 ou M2 a un et un seul méta-élément (méta-noeud ou méta-lien) auquel il est rattaché, une fois défini, par un lien de type *meta*. Les liens de type

meta permettent alors d'indiquer la nature de chaque élément représenté au niveau M3 ou M2. La définition de *meta* est présentée à la Figure 5. Chaque méta-noeud (ou méta-lien) doit être conforme à *NodeType* (ou *LinkType*). Donc, les structures possibles pour *meta* déduites de sa structure initiale (Figure 5) sont présentées dans la Figure 6. Un *T* (ou un *R*) doit être associé par un lien de type *meta* à un *NodeType* (ou un *LinkType*). La règle sémantique Règle 4 est à retenir. Alors, les rapports de conformité entre les éléments de base de notre M3 décrits au dessus peuvent être illustrés par la Figure 7.

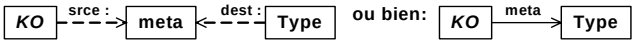


Figure 5. *Définition de meta*

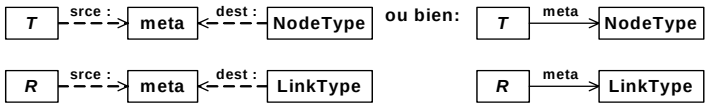


Figure 6. *Structures possibles de meta*

Nous présentons maintenant trois règles qui régissent les éléments de base vus ci-dessus.

Règle 1 : Relation de sous-typage. Si un concept B est lié à A par un lien de sous-typage, chaque instance de B peut se comporter comme instance de A et elle compte parmi les instances de A. Également, si B est un type d'associations, les types unis par B peuvent aussi être unis par A. La relation de sous-typage est transitive et acyclique.

Règle 2 : Restriction de type. Dans un modèle, un type peut être remplacé par un sous-type pour en résulter un autre modèle.

Règle 3 : Sous-typage entre méta-noeuds et sous-typage entre méta-liens. Un méta-noeud (méta-lien) ne peut pas être sous-type d'un méta-lien (méta-noeud).

Règle 4 : Rapport existentiel entre un nœud (lien) et son méta-nœud (méta-lien). Un nœud (respectivement lien) peut exister si et seulement si son méta-nœud (respectivement méta-lien) est défini auparavant au niveau méta. La Figure 7 montre les relations de conformité existant entre les éléments de base de M3

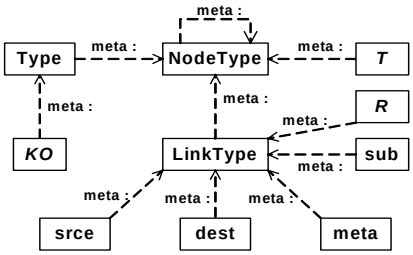


Figure 7. *Éléments de base du méta-métamodèle M3*

4.2. *Autres éléments de M3*

Nous présentons succinctement ci-dessous quelques uns des autres éléments du méta-métamodèle.

globalCard. Les liens de type *globalCard* permettent de représenter des contraintes de cardinalités sur un méta-lien

Language, Value, valueOf, depend. Ces quatre éléments permettent de traiter des valeurs. *Language* et *Value* sont utilisés pour représenter respectivement les langues et les valeurs de l'univers du discours qui servent à représenter les interprétations des éléments. Ces langues et valeurs sont spécifiées et attachées par les liens de type *depend*/*valueOf*.

Model, Metamodel, Structure, IfThenModel, sem, defAs. Parmi les modèles (*Model*), nous distinguons les métamodèles (*MetaModel*), les structures (*Structure*), et les modèles de conditions (*IfThenModel*). *MetaModel* représente l'ensemble des métamodèles. Un modèle se conforme (*sem*) à un et un seul métamodèle. *Structure* vise à représenter, pour les méta-liens, les structures ou bien les modèles de définition qui spécifient comment un méta-lien relie ses éléments. Cela est indiqué par des liens de type *defAs*.

defIn, contain, extend, restrict, infer, result, join, diff, intersection. Ces méta-liens permettent aussi de gérer les modèles et leur contenu. Les éléments sont définis (*defIn*) dans un et un seul modèle. Et un modèle peut contenir (*contain*) plusieurs éléments. Un modèle peut étendre (*extend*) d'autres modèles pour fin de réutilisation des éléments du modèle. Il peut aussi permettre de restreindre (*restrict*) ou inférer (*infer*) d'autres modèles. Les modèles peuvent être le résultat (*result*) d'opérations entre modèles telles que des jointures (*join*), des différences (*diff*) ou des intersections (*intersection*).

Nous détaillons ici l'élément *IfThenModel* car il permet de représenter les règles sémantiques des métamodèles.

IfThenModel sert à contextualiser les préconditions et postconditions dans la représentation des règles sémantiques. Relativement au besoin de représenter les règles qu'un élément doit observer, une règle (*Rule*) est attachée à un modèle de préconditions (*IfThenModel*) et à un modèle de postconditions (*IfThenModel*) respectivement par les liens de type *if* /*then*. Pour le traitement des variables dans les règles, nous avons défini les types *Ref* et *EveryRef*. *Ref* correspond à l'interprétation du quantificateur existentiel alors que *EveryRef* correspond à celle du quantificateur universel. La Figure 8 montre un exemple de représentation de la Règle 4 pour le cas de conformité entre liens et méta-liens. Pour tous les éléments possibles représentés par les variables *Rule3-v1*, *Rule3-v2* et *Rule3-v3* (ce qui sont de type *EveryRef*), s'il y a un lien de type *Rule3-v1* reliant *Rule3-v2* à *Rule3-v3*, il doit exister les éléments représentés par les variables *Rule3-v4*, *Rule3-v5* (ce qui sont de type *Ref*) pour que ces conditions soient satisfaites : (i) *Rule3-v4* et *Rule3-v5* sont respectivement le méta-élément de *Rule3-v2* et celui de *Rule3-v3* (ce qui est représenté par un lien de type *meta* de *Rule3-v2* à *Rule3-v4* et par un autre de *Rule3-v3* à *Rule3-v5*); et (ii) *Rule3-v1* est un méta-lien défini pour relier *Rule3-v4* à *Rule3-v5*.

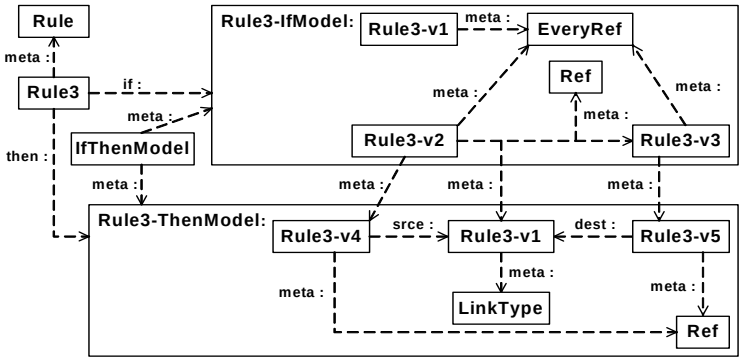


Figure 8. *Exemple de règles*

5. Évaluation du méta-métamodèle

Notre méta-métamodèle (M3) a été défini en réponse aux besoins exprimés dans la section 3. Ce méta-métamodèle permet de représenter des métamodèles de types différents au niveau M2. Pour illustrer cette capacité, les Figure 9, Figure 10, et Figure 11 montrent la représentation de *Marie* dans différents formalismes dont on trouve les éléments nécessaires au niveau M2. Dans ces figures, les liens entre des éléments de niveaux différents sont des liens de conformité. Dans la suite de cette section, nous expliquons chacune des mises en œuvre du méta-métamodèle.

La Figure 9 montre la représentation de *Marie* dans sNets. *Marie* définie dans le modèle *UnModèleM1sNets* au niveau M1 est conforme à *sNetsObject* dans le contexte global, et est également une instance de *Personne* dans le contexte local (ce qui est spécifié par un lien d'instanciation *sNetstype* sur l'axe horizontal). *Personne* est conforme à *sNetsClass*. Dans sNets, *sNetsObject* et *sNetsClass* représentent respectivement l'ensemble de tous les objets et celui de tous les types d'objets; ils sont conformes donc à *NodeType*. Le méta-lien *sNetstype* entre *sNetsObject* et *sNetsClass* est conforme à *LinkType*.

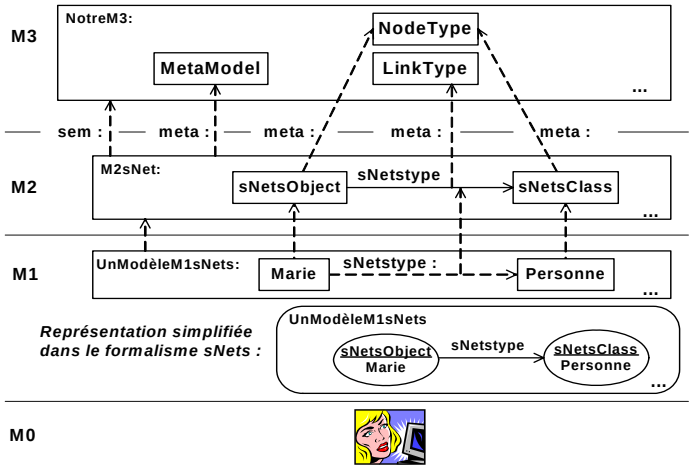


Figure 9. *Marie et sNets*

La représentation de *Marie* dans un formalisme orienté-objet, par exemple UML, est illustrée par la Figure 10. Dans cette figure, *Marie* définie dans le modèle *UnModèleUml* au niveau M1 est conforme à *Instance* dans le contexte global, et est aussi une instance de *Personne* dans le contexte local (ce qui est spécifié par un lien d'instanciation *umltype* sur l'axe horizontal). *Personne* est conforme à *Class*. Dans UML, *Instance* et *Class* représentent respectivement l'ensemble des instances et celui des classes; ils sont conformes donc à *NodeType*. Le méta-lien *umltype* entre *Instance* et *Class* est conforme à *LinkType*.

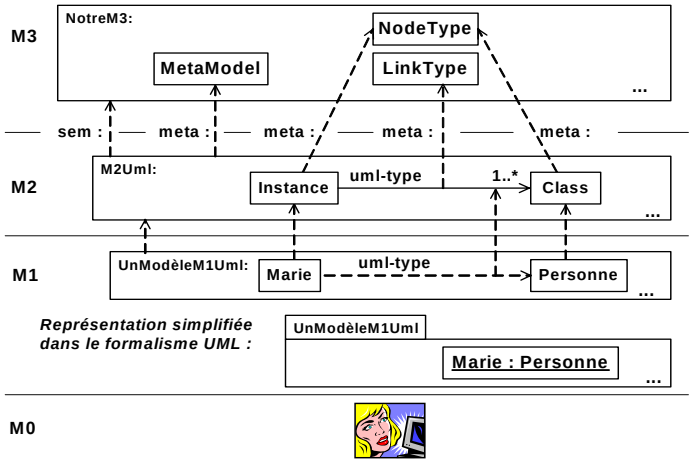


Figure 10. *Marie et UML*

La représentation de *Marie* dans un formalisme utilisé dans le contexte du web sémantique, par exemple RDF/RDFS/OWL, est montrée par la Figure 11. Dans cette figure, *Marie* défini dans une ontologie *UneOntologie* au niveau M1 est conforme à *owlThing* dans le contexte global, et est en outre une instance de *Personne* dans le contexte local (ce qui est spécifié par un lien d'instanciation *rdfstype* sur l'axe horizontal). *Personne* est conforme à *owlClass*. *owlThing* et *owlClass* représentent vis-à-vis l'ensemble des individus et celui des classes; ils sont conformes donc à *NodeType*. Le méta-lien *rdfstype* entre *owlThing* et *owlClass* est conforme à *LinkType*.

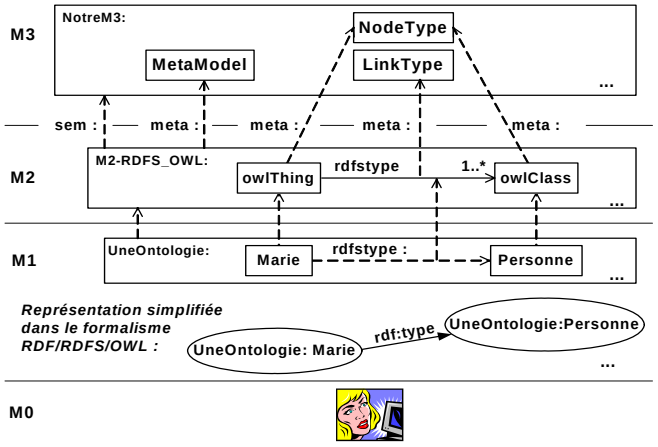


Figure 11. *Marie et RDF/RDFS/OWL*

Nous avons présenté ci-dessus un exemple simple qui illustre la puissance de représentation de notre méta-métamodèle. Pour continuer l'évaluation, nous présentons maintenant une comparaison avec d'autres formalismes

Comme mentionné dans l'introduction, il existe évidemment d'autres candidats pour la représentation de métamodèles : les graphes conceptuels (GCs) (Sowa, 1984; American National Standard, 1999; Gerbé, 2000; Gerbé *et al.*, 2003b), sNets (Lemesle, 2000) basé sur les réseaux sémantiques (Sowa *et al.*, 1991); CDIF (Flatscher, 2002); MOF (OMG, 2003b; OMG, 2004a); XML-XML Schema, RDF-RDFS et OWL[1]. Cependant aucun de ces formalismes ne remplit tous les besoins présentés dans la Section 3. Nous résumons ci-dessous les différents manques identifiés dans (Dinh et al.. 2005).

Les graphes conceptuels et le modèle uniforme (Gerbé, 2000) ne traitent pas explicitement des modèles et de leurs relations. De plus, l'implémentation des GCs est très difficile, plusieurs problèmes restent encore ouverts dont l'interprétation sémantique, les projections et l'opérateur de «matching» dans les GCs.

Le métamodèle de sNets proposé dans (Lemesle, 2000) ne supporte pas l'héritage entre méta-relations (méta-liens) ni les contraintes de cardinalité minimale d'une méta-relation (méta-lien).

CDIF ne traite pas non plus explicitement des modèles et de leurs relations. Concernant la représentation des relations, il permet des relations binaires entre entités mais ne permet pas lier une relation et une autre relation.

MOF est le plus complet et remplit les besoins 2, 3, 4 mais pas entièrement les Besoin 1, Besoin 5 et Besoin 6. Concernant le Besoin 1, MOF supporte différentes notions de base mais pas entièrement la notion de lien. En fait, MOF dans (OMG, 2003b) définit un lien entre deux objets mais un lien n'est pas vu comme un objet ; MOF dans (OMG, 2004a; OMG, 2006) définit un lien entre deux éléments mais un lien n'est pas vu comme un élément. Ceci indique que MOF ne couvre pas toutes les structures possibles pour un lien. Quant aux Besoin 5 et Besoin 6, MOF ne fait pas de distinction entre la relation de conformité d'un élément à un méta-élément et la relation de conformité d'un modèle à un métamodèle et les représente par la même relation « instanceOf » bien que ces deux relations soient de natures très différentes; il n'explicite pas notre notion de modèles de conditions; et il ne permet pas de représenter différents types de liens entre modèles comme *l'intersection, la différence, l'inférence, la restriction.*

Enfin, les formalismes XML-XML Schema, RDF-RDFS et OWL sont utilisés dans le contexte du Web sémantique (Berners-Lee, 2001). XML et XML Schema sont reconnus pour leur flexibilité d'expression mais aussi pour l'ambiguïté sémantique dans la représentation qui rend difficile le traitement sémantique des

[1] W3C. http://www.w3.org/, 2004. Notre analyse est basée sur les documents de spécifications datés de 2004.

informations. RDF, RDFS et OWL sont des langages développés explicitement pour la représentation sémantique de l'information sur le Web. Cependant, ils ne supportent pas tous nos besoins pour M3. Par exemple, ils n'explicitent pas différents types de modèles cités dans le Besoin 5 tels que les notions de métamodèles, modèles structurels, modèles de conditions. Ils ne font pas de distinctions entre les notions : conformité entre éléments de différents niveaux, conformité sémantique entre modèles, et instanciation locale « types-instances ». Ils ne considèrent pas non plus les différents liens entre modèles pour la gestion de modèles tels que l'intersection, la différence, l'inférence, la restriction, etc.

6. Conclusion et Travaux futurs

Les travaux sur la gestion de modèles sont applicables dans divers domaines dont la gestion de connaissances qui nous tient particulièrement à cœur. La représentation de modèles y est un axe de recherche essentiel. Comme les formalismes de modélisation couramment utilisés ne rencontrent pas toutes les exigences pour la représentation de modèles, nous avons spécifié un nouveau formalisme qui devrait satisfaire tous les besoins de la gestion de connaissances. Ce formalisme est basé sur les réseaux sémantiques pour fin de facilité de mise en œuvre. Il est important de noter que le méta-métamodèle est extensible. En se basant sur le noyau du méta-métamodèle, il est possible de définir et d'ajouter de nouveaux éléments au besoin.

Conformément au méta-métamodèle, un métamodèle (niveau M2) est défini pour répondre aux besoins concernant la représentation du monde réel. Nos travaux portent actuellement à la définition d'un métamodèle permettant la représentation de connaissances. Les travaux porteront ensuite à la validation théorique de ce métamodèle ainsi que la réalisation d'un outil de validation pour le méta-métamodèle et le métamodèle. Nous travaillerons sur des opérations de manipulation de modèles qui constituent également un axe important dans la gestion de modèles de connaissances.

Réferences

Alagic S., Bernstein P.A., «A Model Theory for Generic Schema Management», *Proc. DBPL 2001*, Italy, Vol. 2397/2002, Springer-Verlag Heidelberg, 2001, p. 228–246.

American National Standard, *Conceptual Graph Standard*, 1999.

Bernstein P.A., Levy A.Y., Pottinger R.A., «A Vision for Management of Complex Models», *SIGMOD*, Record 29(4), 2000, p. 55-63.

Bézivin J., Blay M., Bouzeghoub M., Estublier J., Favre J.-M., Rapport de synthèse, Action spécifique CNRS sur l'Ingénierie Dirigée par les Modèles, janvier, 2005.

Bézivin J., Gerbé O., «Towards a Precise Definition of the OMG/MDA framework», *Proceedings of the 16th International Conference on Automated Software Engineering*, 2001.

Berners-Lee T., Hendler J. , Lassila O., The Semantic Web, Scientific American, May 2001.

Chen P., «The Entity-Relationship Model -- Towards a Unified View of Data», *ACM Transactions on Database systems*, Vol. 1(1), 1976, p. 9-36.

Connolly D., Harmelen F.v., Horrocks I., McGuinness D.L., Patel-Schneider P.F., Stein L.A., *DAML+OIL (March 2001) Reference Description*, W3C Note, 2001.

Dinh L-A, Gerbé O., Houari S., «Gestion de modèles: définitions, besoins et revue de littérature», *Actes des premières journées sur l'Ingénierie Dirigée par les Modèles (IDM05)*, Paris, France, Juin-Juillet 2005, p. 1–15.

Dinh T.-L.-A., Métamodèle pour la gestion des modèles, Partie orale de l'examen pré-doctoral, Université de Montréal, 2003.

Dinh T.-L.-A., Gerbé O., «A Metamodel for Knowledge Management», *RIVF'04 - International Conference of French-Speaking or Vietnamese Computer Scientists*, Hanoi, Vietnam, 2004, p. 107-112.

Flatscher R.G., «Metamodeling in EIA/CDIF—Meta-Metamodel and Metamodels», *ACM Trans. on Modeling and Computer Simulation (TOMACS)*, Vol. 12(4), 2002, p. 322–342.

Gerbé O., Un modèle uniforme pour la modélisation et la métamodélisation d'une mémoire d'entreprise, Thèse de doctorat, Université de Montréal, 2000.

Gerbé O., Kerhervé B., Srinivasan U., «Model Operations for Quality-Driven Multimedia Delivery», *In Contributions to ICCS 2003*, 2003.

Gerbé O., Mineau G., Keller R., La métamodélisation et les graphes conceptuels, Cahier du GReSI no 03-01, HEC Montréal, 2003.

Girard S., Favre J-M, Muller P-A, Blanc X. editors, *Actes des premières journées sur l'Ingénierie Dirigée par les Modèles (IDM05)*, Paris, France, Juin-Juillet 2005.

Jézéquel J.-M., Gérard S., Mraidha C., Baudry B., Approche unificatrice par les modèles, Action spécifique CNRS sur l'Ingénierie Dirigée par les Modèles, janvier 2005.

Kerhervé B., Gerbé O., «Model Management for Quality of Service Support», *Proc. of the 14th Int. Conf. on Soft. & Syst. Engineering and their Applications*, Vol. 1, France, 2001.

Lemesle R., Techniques de Modélisation et de Méta-modélisation, Thèse de Doctorat, Université de Nantes, 2000.

Melnik S., Generic Model Management, Ph.D Thesis, Lecture Notes in Computer Science, Springer, 2004.

OMG, MDA Guide Version 1.0.1. Joaquin Miller & Jishnu Mukerji ed., 2003a.

OMG, Meta Object Facility (MOF) 2.0 Core Specification, OMG Adopted Specification, ptc/03-10-04, 2003b.

OMG, Meta Object Facility (MOF) 2.0 Core Specification, OMG Adopted Specification, ptc/04-10-15, 2004a.

OMG, Meta Object Facility (MOF) 2.0 Core Specification, OMG Adopted Specification, formal/06-01-01, 2006.

OMG., Unified Modeling Language: Infrastructure, Version 2.0, OMG Adopted Specification, ptc/04-10-14, 2004b.

OMG., Unified Modeling Language: Superstructure, Version 2.0, OMG Adopted Specification, ptc/05-07-04, 2005.

Revault N., Sahraoui H.A., Blain G., Perrot J.-F., «A Metamodeling technique: The MétaGen system», *In Tools 16: Tools Europe'95*, Prentice Hall, Versailles, France, Mar., 1995, p. 127-139 http://www-poleia.lip6.fr/~revault/papers/TOOLSEur95-RSBP.pdf.

Sahraoui H.A, Application de la méta-modélisation à la génération d'outils de conception et de mise en oeuvre de bases de données, Thèse de Doctorat, Université P. et M. Curie (Paris 6), Paris, France, 1995.

Schneider D., Modélisation de la démarche du décideur politique dans la perspective de l'intelligence artificielle. Thèse de doctorat, Université de Genéve, 1994.

Sowa J.F., Conceptual Structures – Information processing in mind and machine, Addison wesley 14472, 1984.

Sowa J.F., Borgida A., Principles of Semantic Networks: Explorations in the Representation of Knowledge, Morgan Kaufmann Publishers, San Mateo, CA, 1991.

Stachowiak, H., Gedanken zu einer allgemeinen Theorie der Modelle; Studium Generale, Springer, 1965.

Rétro-ingénierie dirigée par les métamodèles

Concepts, Méthodes et Outils

Jean-Marie Favre* — Jonathan Musset**

**Université Joseph-Fourier*
Grenoble, France
http://www-adele.imag.fr/~jmfavre

***Obeo*
95 route de gachet
44307 Nantes, France
http://www.obeo.fr/

RÉSUMÉ. *L'Ingénierie Dirigée par les Modèles est un thème en pleine expansion aussi bien dans le monde académique que dans le monde industriel. Bien que l'on puisse imaginer un futur basé sur l'utilisation systématique de modèles, il n'en reste pas moins que les pratiques industrielles sont centrées sur le code. Le succès de l'IDM sera non seulement lié à la prise en compte des nouveaux développements mais aussi et surtout à la prise en compte des logiciels existants voire patrimoniaux. Cet article montre que l'IDM est en fait extrêmement bien adapté à cette problématique. La notion de métamodèle, trop souvent négligée dans le domaine de la rétro-ingénierie, se révèle en fait être un point clé. La rétro-ingénierie dirigée par les métamodèles est une piste extrêmement prometteuse. Cette thématique est abordée dans cet article selon trois axes. (1) D'un point de vue conceptuel, on montre que les concepts fondamentaux de l'IDM et de la rétro-ingénierie se révèlent en fait être les mêmes. (2) D'un point de vue méthodologique, cet article montre que la notion de métamodèles est au coeur des processus de ré ingénierie globale. Le processus CacOph*oN*y est brièvement décrit à titre d'exemple. (3) D'un point de vue outils, on montre que les technologies de l'IDM s'appliquent à la rétro-ingénierie et à l'évolution des logiciels existants. A titre d'illustration, un exemple de cartographie dirigée par les métamodèles est présenté en utilisant l'environnement Obeo Agile Reverse basé sur Eclipse et EMF.*

MOTS-CLÉS : *Ingénierie Dirigée par les Modèles, IDM, Rétro-ingéniérie, Evolution, Maintenance, Métamodèle, Transformation, Analyseur, Cartographie, Urbanisation, Logiciels Patrimoniaux.*

1. Introduction

Les discours futuristes prévoyant la "disparition du code" et la production automatique de logiciels à partir de modèles abstraits ne doivent pas faire oublier l'état de la pratique en génie logiciel : plus d'un demi-siècle après les débuts de l'informatique, l'industrie du logiciel est toujours résolument centrée sur le code... Bien que l'*Ingénierie Dirigée par les Modèles* (IDM, MDE en anglais) soit une approche fort prometteuse [1][2][3][4], négliger l'existant et ignorer les problématiques liées à l'évolution des logiciels est sans doute le meilleur moyen de faire avorter la "révolution" que certains prédisent. D'un point de vue pragmatique nous pensons que l'IDM devrait plutôt être vue comme une évolution plus qu'une révolution. Alors que la très grande majorité des travaux existant se concentrent presque exclusivement sur la *génération de code* à partir de modèles, nous pensons indispensable d'intégrer dans les démarches IDM des fonctionnalités de *rétro-ingénierie* [5][6] permettant d'extraire des modèles à partir de code existant. Ingénierie et rétro-ingénierie doivent être vues comme deux facettes indissociables de tout processus d'évolution. Le développement n'est pas le point dur du génie logiciel, le problème est l'évolution [8][9][10].

Pour bon nombre d'informaticiens l'Ingénierie Dirigée par les Modèles a une connotation "technologie de pointe" voire futuriste, alors que Cobol a une connotation passéiste. En dépit des qualités certaines des langages mûrs et matures, ces mythes sont particulièrement répandus dans le monde académique [11]. L'association des termes Cobol et métamodèle est souvent perçue comme incongrue, notamment par les néophytes. Il n'en est pourtant rien. Bien au contraire.

D'un point de vue très pragmatique, c'est justement lorsque les langages et les technologies sont difficiles à comprendre que la plus-value de les formaliser via des métamodèles est importante [14].

Alors qu'un nombre significatif de travaux de recherche visent à définir des métamodèles pour des langages ou des technologies dont l'avenir est hypothétique, cet article s'intéresse à l'application des principes de l'IDM aux logiciels existants ; et ce tel qu'on les trouve dans l'industrie, quel que soit le niveau de maturité ou d'obsolescence des langages utilisés.

Pour fixer les idées indiquons que les applications logicielles qui sont à la base de la problématique traitée dans cet article peuvent typiquement dépasser le million de lignes de code, sont écrites dans des langages provenant de générations différentes, sont basés sur des technologies parfois ad hoc ou propriétaires. La durée de vie de tels logiciels se compte typiquement en décennies, voire en fraction de siècles [12], tout comme d'ailleurs la durée de vie de langages éprouvés comme Cobol, Fortran ou le langage C. Rappelons que selon diverses estimations environ 80% du volume de code actuellement en exploitation sur notre planète serait écrits dans des langages non structurés. Et c'est bien sur cette même planète que les approches de type MDE sont censées se développer … [3].

Selon nous le futur de l'IDM, et notamment son adoption à une large échelle, résident tout autant dans la prise en compte des *logiciels patrimoniaux* que dans le développement "from-scratch" de logiciels, fussent ils basés sur les dernières technologies à la mode. A ce propos il est ici essentiel de rappeler que le terme *patrimoine* n'a pas une connotation négative dans le langage courant. Après tout, comme le fait remarquer Parnas dans son fameux article

"Software Aging" [8], ce sont uniquement les logiciels qui ont du succès qui atteignent une maturité et un âge avancé. Beaucoup de logiciels ne sont au contraire pas suffisamment utiles pour que l'on décide de les maintenir et de les faire évoluer.

Seul les logiciels utiles et les langages utiles traversent les décennies.

Par ailleurs il a été montré que le terme patrimonial, traditionnellement associé à des logiciels écrits en Cobol, s'appliquait en fait à toutes les générations de paradigmes et de technologies y compris ceux considérés comme "modernes" [13]. La *rétro-ingénierie des logiciels à objets* est par exemple devenu un thème important au cours des dernières années [10]. Remarquons aussi que les méthodes de développement agiles intègrent au sein même de tout processus de développement des activités de refactorisation (refactoring en anglais), Les techniques de restructuration [5], que l'on croyait jusqu'alors réservées aux logiciels anciens, intègrent ainsi le paysage des paradigmes dits modernes (p.e. l'objet), voire des paradigmes émergents. Il devrait être ainsi naturel de prévoir dès à présent les problématiques de rétro-ingénierie de logiciels à composants, de logiciels à services, de logiciels à aspects, etc. En fait, l'apparition d'un nouveau paradigme ou de nouveaux langages donne invariablement naissance à une problématique de rétro-ingénierie adaptée à ce paradigme.

Par exemple, dès la fin du millénaire précédent (notons le changement d'échelle de temps préconisé dans [12]), nous avons mis à jour l'importance de la *rétro-ingénierie des logiciels à composants*, et ce dans le contexte de la société Dassault Systèmes, le leader mondial du CAD/CAM et l'un des pionniers avec Microsoft du paradigme à composants. Nous avons développé dans ce contexte des outils de rétro-ingénierie pouvant servir aussi bien dans une problématique de développement que d'évolution [14], et ce bien avant que la vague des composants n'intègre la longue liste des modes informatiques [13].

Notons, et nous verrons que cela est particulièrement important pour bien comprendre la problématique à laquelle on s'intéresse dans cet article, que dans le cas cité précédemment la technologie utilisée était propriétaire, car développée en interne, par et pour Dassault Systèmes. Nous avons alors dû dans ce contexte extraire un *métamodèle dédié*, ou langage dédié (*domain specific language*, DSL en anglais). Ces concepts seront décrits plus précisément dans cet article. Pour l'instant notons simplement que si l'on se réfère à des technologies plus répandues, les besoins en terme de rétro-ingénierie ne diminuent pas. Il est par exemple fort probable que les logiciels basés sur des technologies "classiques" telles que les EJBs soient considérés dans quelques années comme des applications patrimoniales.

Tout environnement IDM devrait intégrer à la fois des techniques de génération de code à partir de modèle mais aussi des techniques de rétro-ingénierie pour extraire des modèles à partir du code.

La littérature concernant les approches génératives et l'IDM est particulièrement bien fournie [3]. D'autre part la communauté travaillant sur le thème de la rétro-ingénierie est déjà ancienne [5][6]. Par contre l'intersection entre les communautés rétro-ingénierie et IDM est relativement dépourvue, en tout cas au regard des enjeux industriels et scientifiques correspondants.

Cet article est un manifeste pour la *rétro-ingénierie dirigée par les métamodèles*, en tant que problématique industrielle majeure, mais aussi en temps qu'axe de recherche. Cet article établit un bref panorama de la rétro-ingénierie dirigée par les métamodèles en considérant successivement les concepts, les méthodes et les outils. En ce qui concerne les méthodes et les outils, nous n'avons pas cherché à établir un état de l'art exhaustif. Nous avons choisi au contraire d'illustrer chacun des différents aspects abordés à partir des techniques que nous avons développées au cours des dernières années. Bien évidemment des références vers d'autres travaux sont fournis tout au long de l'article.

Le reste de cet article est structuré comme suit. La section 2 ("concepts") montre que les bases de la rétro-ingénierie et de l'IDM sont en fait les mêmes. La section 3 ("méthodes") montre que les métamodèles peuvent jouer un rôle central dans les processus de rétro-ingénierie. La méthode *CacOpho*N*y* est brièvement présentée à titre d'illustration. La section 4 ("outils") décrit un scénario simple dans laquelle on cherche à obtenir une cartographie d'une application patrimoniale écrite en C et en Cobol. L'environnement Acceleo d'Obeo est présenté à titre d'exemple. Finalement la section 7 conclut et présente les nombreuses perspectives qu'ouvre ce travail.

2. CONCEPTS : Rétro-ingénierie et IDM - des bases conceptuelles identiques

Pour beaucoup d'informaticiens l'IDM et plus ou moins synonyme de génération de code. Il s'agit d'une vision bien réductrice car l'IDM a un potentiel suffisamment large pour couvrir tout le spectre du cycle de vie [2], et ceci inclut la rétro-ingénierie. En fait, comme nous allons le voir l'IDM et la rétro-ingénierie sont basés sur les mêmes concepts fondamentaux.

2.1. *Définitions et concepts fondamentaux de l'IDM*

Bien que les notions de *modèles*, de *métamodèles* et surtout de transformations restent à définir de manière précise, il existe toutefois un relatif consensus sur le fait que ces notions forment les piliers de l'IDM [1][2]. Rappelons ci-dessous deux définitions. Le lecteur intéressé par un ensemble plus vaste de définitions pourra se référer à la section glossaire de planetmde.org [3] ainsi qu'à nos travaux sur la "mégamodélisation" (chapitre 2 de [2], [19]).

Modèle	*A **model** is a simplification of a **system** built with an intended goal in mind. The model should be able to answer questions in place of the actual system. [18]*
Métamodèle	*A **metamodel** is a model that defines the **language** for expressions a **model**. (MOF Standard, Version 1.4)*

2.2. *Définitions et concepts fondamentaux de la rétro-ingénierie*

Après des premières années caractérisées par une terminologie foisonnante [6], les bases conceptuelles de la rétro-ingénierie et le vocable associé ont été établis en 1990 par Chikofsky et Cross dans l'article "Reverse Engineering and Design Recovery: a Taxonomy" [11] (voir table 1). Cette taxonomie fait toujours référence et c'est donc sur ces définitions qu'est basé cet article. Comme le montre la Figure 1, extraite de [5] la taxonomie est basée sur une classification des transformations qu'il est possible d'appliquer au logiciel. Le concept de *transformation* est ainsi au cœur de la rétro-ingénierie. La légende originelle de la Figure 1 indique : "la rétro-ingénierie et les processus associés sont des transformations inter ou intra

niveaux d'abstraction". Quelques années plus tard, Arnold raffine cette taxonomie en fournissant "une procédure décisionnelle pour classifier les *transformations du logiciel*" [6].

Ingénierie directe *Forward engineering*	*Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system*
Rétro-ingénierie *Reverse engineering*	*Reverse engineering is the process of analyzing a subject system with two goals in mind: (1) to identify the system's components and their interrelationships; and, (2) to create representations of the system in another form or at a higher level of abstraction*
Redocumentation	*Redocumentation is the creation or revision of alternate views semantically coherent with the examined system*
Rétro-conception *Design recovery*	*Design recovery is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself*
Restructuration *Restructuring*	*Restructuring is the transformation from one representation to another at the same relative abstraction level, while preserving the subject system's external behaviour*
Re-ingénierie *Reengineering*	*Reengineering is the examination of a subject system to reconstitute it in a new form and the subsequent implementation of the new form*

Table 1. Taxonomie de Chikofsky and Cross.

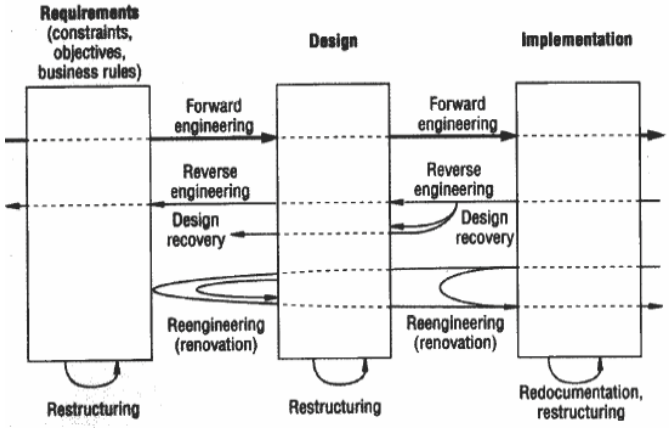


Figure 1. Classification des transformations. Figure extraite de [5]

La notion de transformation est donc clairement au cœur de la rétro-ingénierie, mais qu'en est-il de la notion de *modèle* ? Ce mot n'apparaît pas dans l'article originel [5]. Ce concept est pourtant bien présent, notamment dans la définition du terme rétro-ingénierie puisqu'il s'agit "d'analyser un système" et de créer des "représentations de ce système". On retrouve ici la relation de représentation (RepresentationDe, μ) qui caractérise la notion de modèle [2][19]. En fait, l'article fondateur de Chikofsky et Cross pourrait être réécrit sans en changer le sens en remplaçant les mots "représentation", "vue" et "abstraction" par "modèle".

IDM et rétro-ingénierie partagent les mêmes fondements conceptuels.
Ces deux démarches sont donc vouées à être intégrées.

Le lecteur attentif aura sans doute remarqué que parmi les *trois* piliers de l'IDM, on retrouve explicitement dans le domaine de la rétro-ingéniérie que deux notions *modèle* et *transformation*. La notion de métamodèle a tout simplement été oubliée dans la description des fondations de la rétro-ingénierie. Quinze ans après, ce papier vise à combler ce manque

en rendant aux métamodèles la place qu'ils méritent. Mais tout d'abord continuons avec la description des fondations techniques de la rétro-ingénierie.

2.3. *Architecture des outils de rétro-ingénierie*

Au cours des années 90, de très nombreux outils de rétro-ingénierie ont été proposés. Arnold a fait remarquer que tous ces outils partageaient globalement l'architecture présentée dans la Figure 2. A gauche on trouve les artefacts logiciels à partir desquels sont extraites les informations. Dans la terminologie de l'IDM il s'agit du "système étudié" (cf la section 2.2). Les vues produites à droite peuvent être présentées aux ingénieurs sous de multiples formes. Ce sont des "modèles" du logiciel. La base d'information que l'on trouve au centre peut également être considérée comme un modèle du logiciel.

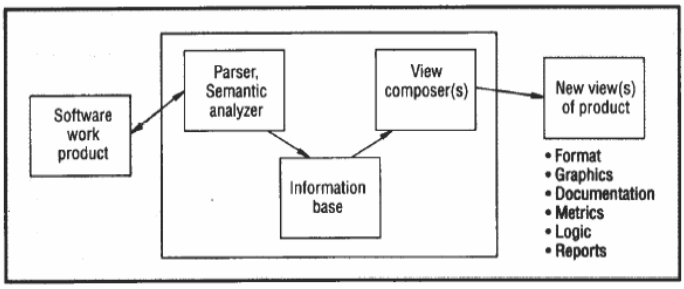


Figure 2. Architecture des outils de rétro-ingénierie [6]

Au point de vue des techniques classiques de rétro-ingénierie on peut bien évidemment citer les techniques d'analyse du logiciel (partie gauche du diagramme), mais aussi les techniques de *visualisation* (partie droite) qui sont alors particulièrement adaptées et l'on souhaite typiquement obtenir des "cartographies" qui sont des modèles visuels des applications logicielles existantes. Dans le monde de la rétro-ingénierie, on parle souvent d'environnement d'*exploration des logiciels* (e.g. [20]), car il s'agit de "naviguer" d'un modèle à l'autre et ce de manière interactive.

L'un des défi auquel les environnements de rétro-ingénierie doivent faire fasse consiste à prendre en compte la multiplicité des sources d'informations disponibles. Cet aspect n'a hélas pas été clairement identifié dans les travaux fondateurs en rétro-ingénierie [5][6]. Par exemple remarquons qu'une et une seule boite est présente à gauche de la Figure 2. Bien que cette boite soit à juste titre appelée "produit logiciel", elle a été pendant très longtemps assimilée à tort au code source, qui n'est en fait qu'un aspect particulier du logiciel.

2.4. *Défi n° 1 : rendre flexible les outils de rétro-ingénierie*

La première génération d'outils de rétro-ingénierie correspondait à des outils construits de manière monolithique et "câblée" centrée sur une fonctionnalité précise. Ces outils étaient typiquement liés à un langage de programmation unique, proposaient un seul type d'analyse et un seul type de vue. C'est le cas par exemple des outils permettant à partir de programmes C d'extraire un graphe d'appels entre fonctions et d'afficher cela sous forme graphique. Ces solutions ad hoc, câblées et monofonctionnelles ne sont plus suffisantes dans le contexte

actuel. On assiste à une multiplication (1) des sources de données en entrée, mais aussi (2) des types de vues à produire en sortie. Chacun de ces deux points est étudié ci-dessous.

La problématique liée à la *multiplicité des sources de données en entrée* dépasse largement l'aspect multi-langages au sens classique, même si celui-ci a été catalogué comme l'un des points difficiles mais essentiel à résoudre en pratique [21]. Désormais il s'agit en effet de prendre en compte une très large gamme d'artefacts logiciels qui inclut par exemple code source, bytecode, binaires, librairies, traces, fichiers de configurations, rapports de bug, etc.

Les environnements de rétro-ingénierie doivent pouvoir extraire des informations provenant de sources d'informations très variées.

Prendre en compte la *multiplicité des vues à produire en sortie* est également un défi pour les outils de rétro-ingénierie. Cet aspect a été identifié plus clairement dans les travaux fondateurs comme le montre la partie droite de la Figure 2. Par contre au cours des 15 années qui ont suivi, la compréhension de ce qu'est un logiciel a grandement évolué. Aujourd'hui le standard IEEE 1471 sur l'architecture des logiciels [22] définit clairement que les différents *intervenants* (stakeholder) d'un projet logiciel devraient pouvoir considérer le logiciel selon différents *points de vues* (view points) en fonction de leur métier et de leurs compétences. Il s'agit là clairement de l'application du principe de *séparations des préoccupations*, qui est à la base aussi de l'IDM. En fait les bases conceptuelles du standard IEEE 1471, de l'IDM et de la rétro-ingénierie sont très proches. Dans le standard IEEE 1471 le terme *vue* correspond à la notion de *modèle*, alors que le terme *point de vue* correspond à la notion de métamodèle [23].

Les environnements de rétro-ingénierie doivent pouvoir générer un ensemble de vues variées et adaptées à chaque type d'intervenant.

2.5. *Défi n° 2 : rendre agiles les processus de rétro-ingénierie*

La plupart des outils classiques de rétro-ingénierie disponibles étant "câblés", ils n'offrent que très peu de souplesse. Sachant qu'ils ne couvrent généralement que certaines fonctionnalités, l'une des difficultés récurrentes des projets de ré ingénierie consiste à trouver une manière de couvrir tous les besoins du projet en intégrant autant que faire se peut les outils dont on dispose. Ainsi l'interopérabilité des outils de rétro-ingénierie est devenue un thème de recherche important. L'une des approches les plus courantes consiste à faire communiquer les différents outils via l'import et l'export de données sous la forme de fichiers écrits dans des standards adaptés à la représentation de graphes, tel GXL. Cette approche offre bien peu de souplesse. Non seulement il est nécessaire d'écrire des "wrappers" permettant de faire la correspondance entre les structures internes manipulées par les outils et les formats externes, mais en plus ces solutions sont basées sur l'échange permanent de gros volumes de données entre outils (typiquement via des fichiers XML). Une telle solution ne permet pas non plus l'intégration fine des outils dans les environnements de programmation existants. L'ajout de nouvelles fonctionnalités est également difficile car il est nécessaire de posséder de nombreuses expertises techniques : analyse syntaxique, transformations vers des formats d'échanges, interface homme-machine, visualisation du logiciel, etc. Toutes ces difficultés résultent dans des processus de rétro-ingénierie lourds et peu flexibles car entièrement contraints par les outils existants. Développer de nouveaux outils pour prendre en

compte des technologies obsolètes ou propriétaires a un coût souvent considéré comme rédhibitoire. En fait, de part la nature exploratoire des processus de rétro-ingénierie, l'un des défis qu'il est nécessaire de surmonter est de minimiser ces coûts de développement de sorte que l'on puisse adapter les outils "à la demande" au cours du déroulement de chaque projet de rétro-ingénierie.

L'un des défis à surmonter est de pouvoir passer de processus de rétro-ingénierie rigides et figés à des processus de *rétro-ingénierie agile*, c'est-à-dire dans laquelle la flexibilité est essentielle et où les outils sont développés à la demande au fur et à mesure du processus.

2.6. Synthèse : vers la rétro-ingénierie dirigée par les métamodèles

En résumé, les nouveaux défis auxquels sont confrontés les projets de rétro-ingénierie sont d'une part (1) de prendre en compte la multiplicité des sources de données et des vues à produire ; et d'autre part (2) de pouvoir adapter processus et outils "à la demande" et ce afin de prendre en compte la nature exploratoire de la rétro-ingénierie.

Les concepts identifiés dans les travaux fondateurs de la discipline ne se sont hélas pas avérés suffisants pour surmonter ces défis : globalement les outils existants sont trop rigides. Et c'est justement là que l'Ingénierie Dirigée par les Modèles intervient. Si l'on compare les concepts de l'IDM (section 2.1, modèle, métamodèle, transformation), et les concepts de la rétro-ingénierie (section 2.2, modèle et transformation), on note que le grand absent est le concept de *métamodèle*. L'article de Chikofsky et de Cross n'y fait d'ailleurs à aucun moment référence. Pourtant il est simple, a posteriori, de voir que ce concept à la base de l'IDM, permet d'apporter la flexibilité nécessaire. Par exemple, si l'on considère l'architecture des outils de rétro-ingénierie (Figure 2), on constate qu'à chaque fois qu'un modèle est présenté, le métamodèle correspondant devrait être explicité, ce qui permettrait d'utiliser des outils génériques de transformations de modèles. C'est là la leçon de l'Ingénierie Dirigée par les Modèles, qu'il s'agit d'appliquer dans le contexte de la rétro-ingénierie.

Nous pensons que le concept de métamodèle est le pilier manquant de la rétro-ingénierie, et qu'il est possible de revisiter toutes les techniques de rétro-ingénierie proposées au cours des dernières décennies à la lumière de ce concept. Nous parlerons alors de *rétro-ingénierie dirigée par les métamodèles.*

3. METHODES : Exemple —Le processus *CacOpho*N*y*

Dans la section précédente l'intersection entre IDM et rétro-ingénierie a été considérée au niveau *conceptuel*. Cette section considère l'aspect *méthode* et montre que la notion de métamodèle peut réellement être au cœur des processus de rétro-ingénierie. Pour illustrer cet aspect nous avons choisi de présenter brièvement la démarche *CacOpho*N*y* qui résume notre expérience acquise au cours des dernières années dans des contextes tels que notre collaboration avec la société Dassault Système. Le lecteur se rapportera par exemple à [14] ou [24] pour avoir plus de détails sur l'utilisation de métamodèles dans le cadre de cette

collaboration industrielle. L'objectif ici est simplement de montrer un exemple de processus de "rétro-ingénierie dirigée par les métamodèles".

Supposons que l'on doive lancer un nouveau projet de rétro-ingénierie dans une grande entreprise. Contrairement à une approche classique, la caractéristique principale de notre solution consiste à rendre explicites les métamodèles utilisés implicitement au sein de l'entreprise considérée, autrement dit à modéliser le savoir faire de l'entreprise en terme de génie logiciel. Cette démarche s'inscrit donc réellement dans le cadre de l'IDM. Notons qu'indépendamment de cette solution, le problème à traiter est répertorié sous le nom de "View Set Scenario" dans le catalogue défini par le Software Engineering Institute (SEI) [22]. Le processus *CacOph*oN*y* a été défini pour résoudre ce problème [21]. Entièrement basé sur la notion de métamodèle, il propose une méthode globale pour définir un environnement de rétro-ingénierie adapté aux besoins spécifiques des entreprises.

La difficulté que cherche à résoudre le processus *CacOph*oN*y* consiste à construire un environnement de rétro-ingénierie adapté à une situation spécifique.

On ne saurait trop insister sur l'énoncé ci-dessus et sur le fait que le point dur auquel on s'intéresse est la *construction* d'un environnement, plutôt que sur son *utilisation*. Autrement dit en terme du standard IEEE 1471, on cherche à définir des *points de vues* lors de la construction de l'environnement (alors que l'exécution de cet environnement permettra d'extraire des *vues* à partir d'un logiciel particulier). En termes d'IDM, on travaille donc naturellement au niveau *métamodèle* (M2) et l'on cherche à instrumenter ces métamodèles (ce qui permettra lors de l'exécution de l'environnement d'extraire des *modèles* d'un logiciel particulier).

Soulignons également que la démarche telle qu'elle est présentée ici est taillée pour une grande entreprise. Les concepts identifiés ici proviennent entre autre de notre expérience dans un contexte où plus de 1200 ingénieurs travaillaient en parallèle sur un logiciel de plusieurs millions de lignes de code, formés de plus de 70000 classes C++, de 8000 composants, etc. Bien évidement le processus proposé ici peut être simplifié pour le cas de petites entreprises, mais l'approche ne change pas même si les différents concepts sont moins évidents à discerner dans le cas simple de petites structures. Qui peut le plus, peut le moins.

Par ailleurs, le processus est décrit ci-dessous de manière séquentielle pour bien montrer les différentes activités, mais bien évidemment l'idée est de suivre un processus agile dans lequel chaque étape s'enchevêtre. En particulier la construction de l'environnement et son utilisation se fait en pratique de manière complètement entremêlée ce qui assure le caractère exploratoire de la rétro-ingénierie : on adapte l'outil au fur et à mesure de son utilisation, voire au cours même de son utilisation pour les techniques interactives les plus avancées [27]. Quoi qu'il en soit pour bien différencier les concepts, le processus est décomposé en 4 phases successives découpées en sous phases. Seule l'esquisse du processus est donné ci-dessous. Rappelons que notre objectif ici est uniquement de montrer l'omniprésence de la notion de métamodèle dans toutes les phases de la rétro-ingénierie. Le lecteur se reportera à [21] pour une description de *CacOph*oN*y*.

3.1. *Analyse du domaine (du génie logiciel).*

La première étape consiste à définir quelles sont les entités et relations utilisées. Il s'agit ainsi d'établir une cartographie des concepts de génie logiciel tels qu'ils sont utilisés dans l'entreprise. Il est essentiel de comprendre que l'on ne fait pas référence à une cartographie d'une application (niveau M1), au contraire que l'on se place ici (comme par la suite) au niveau métamodèle, c'est-à-dire au niveau M2. Le résultat attendu est un ensemble de métamodèles précis établis à partir des artefacts concrets de l'entreprise. La figure 3 décompose l'analyse du domaine en trois sous étapes.

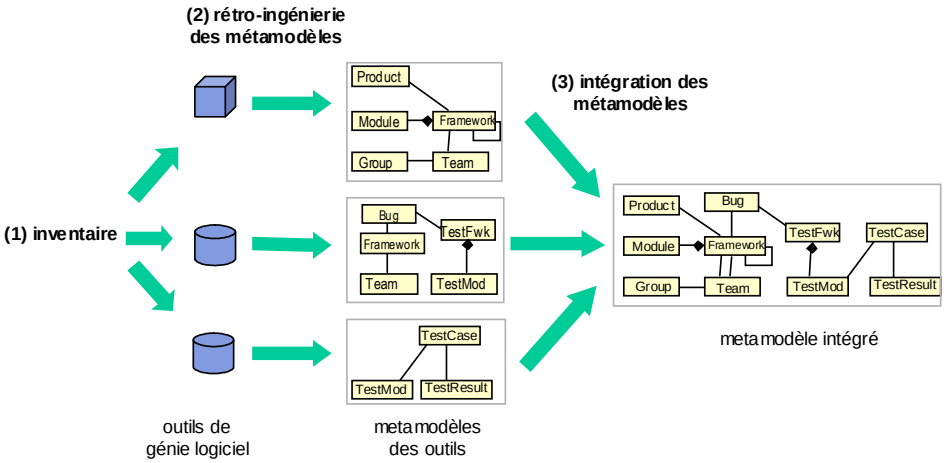


Figure 3. Analyse du domaine

A1) Inventaire des outils de génie logiciel. Il s'agit d'identifier les outils et les artefacts de génie logiciel utilisés afin d'établir une cartographie technologique de l'entreprise comme celle de la figure 3. Cela revient à identifier quelles sont les sources d'informations disponibles. On peut trouver par exemple une base de données de rapports d'incidents, des sources en Cobol et en C, des bases de tests, etc.

A2) Rétro-ingénierie des métamodèles. Il s'agit de reconstituer le métamodèle de chaque outil ou source d'information inventoriée ci-dessus. Dans l'espace technique Gammarware, Lämmel et ses collègues ont identifié cette problématique sous le terme *rétro-ingénierie des grammaires* [25]. Si l'on dispose d'une base de rapports de bugs telle que celle de Bugzilla, à partir du schéma relationnel de cette base, on pourra retrouver un métamodèle conceptuel. De même si l'on dispose d'un compilateur, d'un ensemble de fichiers XML, d'une DTD on pourra retrouver un métamodèle. Tout comme l'ingénierie des langages, la *rétro-ingénierie des langages* et la *rétro-ingénierie des métamodèles* sont des thèmes de recherche émergeants.

A3) Intégration des métamodèles. Il n'est alors pas rare que les outils utilisés donnent lieu à des chevauchements entre métamodèles. La dernière étape dans l'analyse du domaine consiste à intégrer les métamodèles conceptuels des différents outils. L'intégration et la composition de métamodèles sont des thèmes de recherche importants de l'IDM qui soulèvent autant des problèmes conceptuels que techniques.

3.2. *Analyse des (méta) besoins et spécifications externes.*

Trop souvent la notion de métamodèle est déconnectée des besoins. *CacOph*oNy propose de relier ceux-ci au métamodèle global en utilisant la technique traditionnelle des cas d'utilisation, appliquée au niveau méta (M2) et non pas au niveau traditionnel (M1).

(B1) Identification des (méta)acteurs. Il s'agit des utilisateurs des langages de génie logiciel, et non pas les utilisateurs finaux de l'application développée.

(B2) Identification des (méta) cas d'utilisation. Il s'agit d'identifier les tâches réalisées par les différents intervenants de l'entreprise.

(B3) spécification de l'environnement de ré-ingénierie. Il s'agit de spécifier l'environnement à construire, et en particulier de déterminer quel sous-ensemble du métamodèle global est approprié pour réaliser chaque cas d'utilisation.

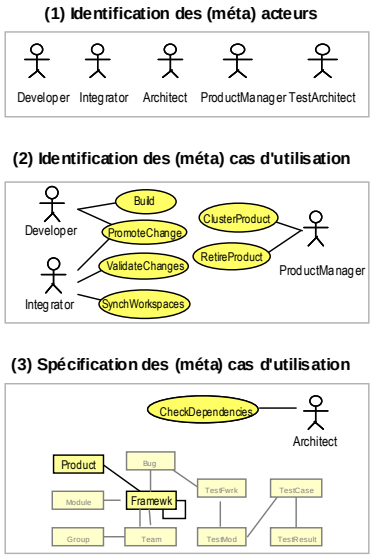


Figure 4. Meta-acteurs/Meta-besoins

3.3. *Intégration et implémentation.*

L'étape suivante consiste à implanter l'environnement spécifié en reproduisant globalement l'architecture présentée dans la Figure 2 (page 56). Il s'agit alors d'intégrer les briques existantes et non pas de les re-développer.

(C1) Extraction de modèles (partie gauche de la Figure 2). Il s'agit d'exploiter la traçabilité établie dans l'étape (A) entre la liste des outils fournis lors de l'inventaire et le métamodèle global. Le but est de trouver quel est l'ensemble des outils à instrumenter pour obtenir les modèles dont on a besoin. Des "wrappers" et/ou des transformations doivent être réalisés pour concrétiser les différentes étapes établies en (A).

*(C2) Présentation des modèles***.** La concrétisation de la partie droite de la Figure 2 peut également se baser sur la réutilisation d'outils et techniques existants, mais aussi sur la paramétrisation d'outils génériques par exemple de visualisation. Des exemples seront donnés dans la section 4.

3.4. *Exécution et tests.*

L'objectif ultime est d'utiliser l'environnement ainsi produit pour supporter le processus d'évolution des applications logicielles développées par l'entreprise. En pratique ceci donne lieu à de nombreux retours et demandes de modification. Comme nous l'avons dit, les étapes présentées ci-dessus s'enchevêtrent en pratique, donnant lieu à des processus dans lesquels l'outil de rétro-ingénierie à construire est développé au fur et à mesure de son utilisation pour répondre aux besoins souvent découverts de manière incrémentale.

3.5. *Synthèse*

Comme le lecteur a pu le constater les métamodèles jouent un rôle central tout au long du processus. Les étapes (A), (B) et (C) se déroulent uniquement au niveau méta (M2) et différentes problématiques ont été identifiées : rétro-ingénierie des métamodèles, transformations de métamodèles, intégration de métamodèles, etc. La terminologie "*rétro-ingénierie dirigée par les métamodèles*" est donc tout à fait justifiée.

4. OUTILS : Exemple — Cartographie avec l'environnement Obeo

Après avoir étudié l'intégration IDM et rétro-ingénierie au niveau conceptuel (section 2), puis au niveau méthode (section 3), cette section se concentre sur l'aspect outil. Nous allons voir que l'utilisation de métamodèles permet d'obtenir des outils de rétro-ingénierie d'une flexibilité inégalée jusqu'alors. Nous illustrerons notre propos à partir d'un cas d'utilisation simple. Supposons que l'on cherche à établir une cartographie d'une application logicielle basée sur des technologies hétérogènes Cobol et C qui, de plus, mélange interface graphique, logique métier et accès aux données. Nous allons voir que la notion de métamodèle et les techniques d'IDM correspondantes permettent de résoudre ce problème de manière agile.

4.1. *Vision globale*

La Figure 5 ci-dessous présente une vision globale du système. Comme on peut le voir, l'architecture utilisée correspond bien à l'architecture classique des outils de rétro-ingénierie (Figure 1, page 55), si ce n'est que l'on peut noter l'omniprésence de métamodèles à chaque étape. On suppose ici que l'application à analyser est formée de programmes Cobol et C à gauche. Le modèle de cartographie en haut à droite montre le résultat à obtenir. On ne décrit par la suite que le chemin décrit en gras, mais bien évidemment d'autres vues peuvent être produites comme le montre la Figure 5. Notons qu'entre les deux extrêmes à droite et à gauche, les transformations utilisent un outil générique de transformation de modèles, ici Obeo Transformer. Tous les métamodèles sont détaillés dans les sections suivantes en parcourant la figure 5 de gauche à droite.

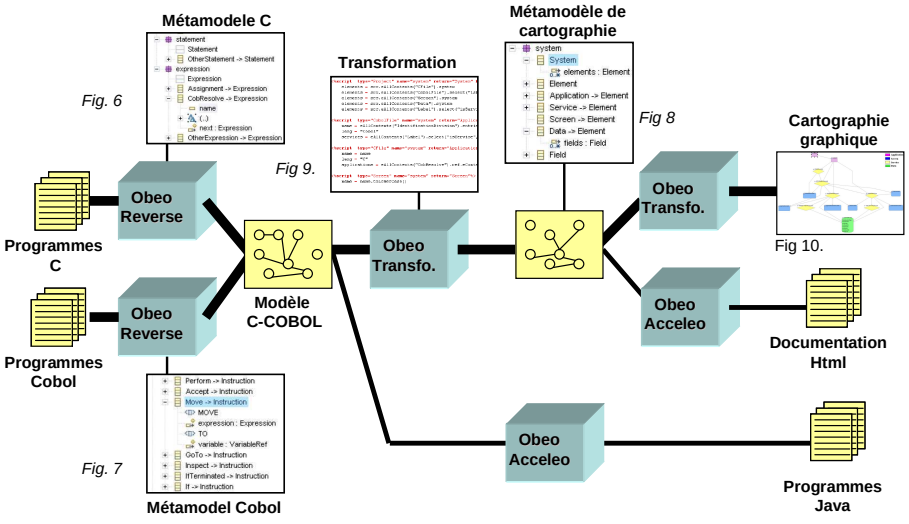


Figure 5. Une chaîne compléte de rétro-ingénierie

4.2. *Extraction des modèles : analyse et métamodèles de code*

La mise au point d'une telle chaîne de rétro-ingénierie commence par la création d'un métamodèle de code qui est décoré avec des informations syntaxiques. La création de ces métamodèles de code grâce à Obeo Reverse permet de se concentrer sur le langage à analyser, car il fusionne les étapes d'analyse syntaxique et de métamodélisation. Un modèle, résultat de l'analyse de technologies hétérogènes, est obtenu à partir : (1) d'un analyseur de surface pour le langage C afin de ne récupérer que les informations macroscopiques de détection des appels aux programmes cobol, autrement dit les appels à une méthode système appelée `cob_resolve` et, (2) d'un analyseur fin pour Cobol détectant précisément tous les aspects du langage nécessaire au cas d'étude considéré.

La figure 6 ci-dessous à gauche montre un extrait du métamodèle de C. L'analyseur de surface C est simple car les informations que l'on veut retirer des fichiers C sont simples. On distingue 2 types d'expression : les expressions « CobResolve » et les autres. Ainsi, le modèle résultant de l'analyse de fichiers C sera facile à exploiter. Le code C ne contient pas de logique métier, ni d'interface graphique. Toute cette connaissance se situe dans le langage Cobol, ce qui explique que l'analyse du code Cobol doit être plus fine. On va chercher à exploiter chaque instruction des sources Cobol. L'extrait du métamodèle pour Cobol présenté dans la figure 7 à droite montre un certain nombre d'instructions du langage et la manière de décorer les éléments du métamodèle avec des informations syntaxiques. On retrouve notamment la structure d'une instruction `MOVE`, permettant d'affecter une expression à une variable : `MOVE <expression> TO <variable>`. Les décorateurs syntaxiques `MOVE` et `TO` sont présents en bleu dans le métamodèle et délimitent les références vers d'autres éléments eux aussi décorés. Cette démarche rappelle l'utilisation de sous règles dans les grammaires classiques.

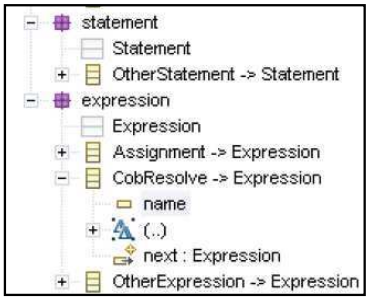


Figure 6. Métamodèle de surface pour le C

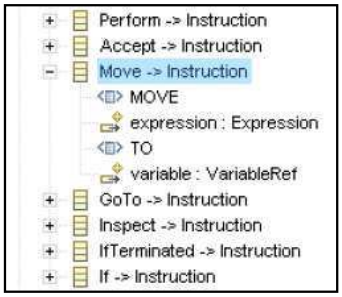


Figure 7. Métamodèle de COBOL

4.2. *Transformations et métamodèle de cartographie*

Les modèles produits par les analyseurs sont proches des technologies sources. Il reste donc à en extraire les informations pertinentes (diagrammes d'architecture, modèles de haut niveau, règles métiers, ...). Cela passe par la définition d'un métamodèle de cartographie. Le métamodèle ci-contre permet de décrire les concepts à représenter, et les dépendances qu'il y a entre eux. Il précise qu'un système est représenté par des éléments d'architecture

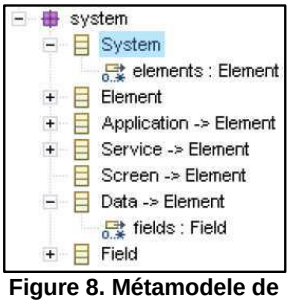


Figure 8. Métamodele de cartographie

qui peuvent être des applications (Application), des services (Service), des écrans (Screen), etc. Bien entendu un tel métamodèle peut être étendu, simplifié, ou adapté en fonction des besoins de chaque intervenant. On peut ainsi réellement outiller des processus agile de rétro-ingénierie. Comme le montre la figure ci-dessous, une transformation permet de peupler ce modèle de cartographie à partir du modèle issu de l'analyse de l'existant. La règle de transformation suivante positionne les informations de création des applications dans le modèle de cartographie, à partir des fichiers Cobol analysés. Par exemple, dans une cartographie, le nom d'une application ne doit pas être basé sur le nom d'un fichier Cobol mais sur l'identifiant du programme défini dans ce fichier. La propriété name de l'application à créer dans la cartographie est donc récupérée en allant chercher la clé PROGRAM-ID dans les profondeurs du code Cobol.

```
<%script  type="CobolFile" name="system" return="Application"%>
    name = eAllContents("IdentificationDivision").entries.select("key","PROGRAM-ID").value
    lang = "Cobol"
    services = eAllContents("Label").select("isService","true").system
```

Fig 9. Extrait de la transformation

4.3. Transformation vers un modèle visuel

Finalement la visualisation graphique du résultat se fait dans un éditeur spécialisé également dirigé par les métamodèles. La figure 9 ci-dessous présente un exemple de modèle produit automatiquement pour une application simple. Ce modèle visuel est conforme au métamodèle de cartographie présenté dans la figure 7. Une des caractéristiques essentielles de la technique de visualisation utilisée est l'aspect intégralement personnalisable. Chaque élément du métamodèle est associé à une représentation graphique différente. Ici, les losanges jaunes sont associés aux services, les cylindres verts aux structures de données, les rectangles bleus aux écrans, etc.

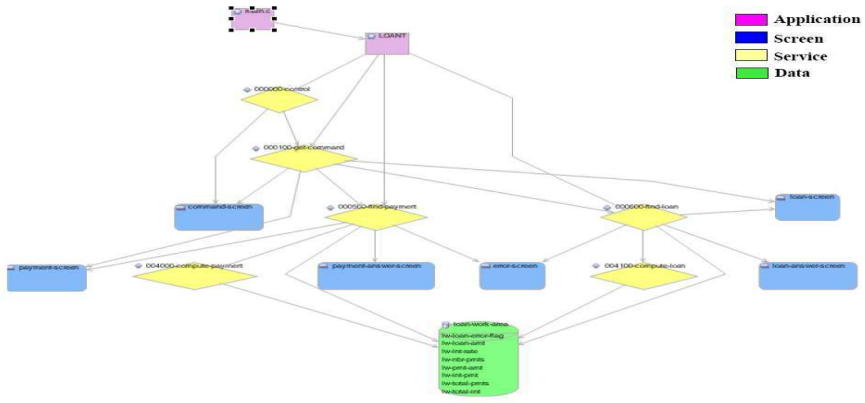


Figure 9. Cartographie résultante

5. Conclusion

En fait il n'existe que peu de différences conceptuelles entre la rétro-ingénierie et la modélisation. Dans un cas on part d'un logiciel existant formé d'artefacts concrets, dans l'autre cas on part de besoins exprimés informellement. Historiquement, le génie logiciel s'est tout d'abord concentré sur la modélisation, ce qui est bien naturel en l'absence d'existant. Mais dans un monde où le logiciel est omni-présent et où la part de l'existant est de plus en plus importante, la tendance devrait peu à peu s'inverser pour arriver à une situation où modélisation et rétro-ingénierie ne seront plus opposées, mais vu au contraire comme deux techniques complémentaires au sein d'un phénomène d'évolution continu.

6. Remerciements

Nous tenons à remercier Stéphane LACRAMPE, Etienne JULIOT, Cédric BRUN de la société Obeo pour leur participation à ce travail.

7. Bibliographie

[1] J. Bézivin, On the Unification Power of Models, Software and System Modeling Journal, 4(2):171--188, 2005

[2] J.M. Favre, J. Estublier, M. Blay, editeurs, *L'Ingénierie Dirigée par les Modèles : au-delà du MDA*, Hermès - Lavoisier, ISBN 2-7462-1213-7, 240 pages, 2006

[3] PlanetMDE, the community web portal for Model Driven Engineering, http://planetmde.org

[4] IEEE Computer, Numéro spécial dédié à l'IDM, Février 2006

[5] E. Chikofsky, J. Cross, *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, Jan. 1990

[6] R. Arnold, *Software Reengineering*, IEEE, ISBN 0-8186-3272-0, 1993

[7] R. Arnold, *On Software Restructuring*, IEEE, ISBN 0-8186-0680-0, 1986

[8] D.L. Parnas. *Software Aging*. ICSE 1994

[9] T. Mens and al. *Challenges in Software Evolution*. IWPSE 2005

[10] S. Demeyer, S. Ducasse, O. Nierstrasz, *Object-Oriented Reengineering Patterns*, Morgan Kaufmann/Elsevier, ISBN 1-55860-639-4, 2003

[11] R. Lammël, *Stop Bashing Cobol*!, Cobol Research Laboratory at the Free University of Amsterdam, http://www.cs.vu.nl/~ralf/Cobol

[12] J.M. Favre. *Languages Evolve Too! Changing the Software Time Scales*. IWPSE 2005

[13] L. Racoon. *Fifty Years of Progress in Software Engineering*. Software Engineering Notes, Vol. 22, n. 51, Jan. 1997

[14] J.M. Favre and al. *Reverse Engineering a Large Component-based Software Product*. CSMR 2001

[15] R. Marvie, L. Duchien, M. Blay, *Les Plateformes d'exécution et l'IDM*, Chapitre 4, [2]

[16] J.M. Favre, J. Bézivin, I. Bull, "*Evolution, rétro-ingénierie et l'IDM : du code aux modèles*", Chapitre 8, [2]

[17] Obeo, *Acceleo, une solution 100% open source pour l'IDM*, http://www.acceleo.org

[18] J. Bézivin, O. Gerbé, *Towards a Precise Definition of the OMG/MDA Framework*, ASE 2001

[19] J.M. Favre, *Towards a Basic Theory to Model Model Driven Engineering*, WiSME 2004, papier disponible à partir de http://www-adele.imag.fr/~jmfavre

[20] J.M. Favre, *GSEE : A Generic Software Engineering Environment*, IWPC 2001

[21] L. O'Brien, C. Stoermer, C. Verhoef, *Software Architecture Reconstruction: Practice Needs and Current Approaches*, SEI Technical Report CMU/SEI-2002-TR-024, 2002

[22] *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE Std 1471-2000

[23] J.M. Favre, *CacOphoNy: Metamodel-Driven Architecture Reconstruction*, WCRE 2004

[24] J. Estublier, J.M. Favre, R. Sanlaville, *An Industrial Experience with Dassault Systèmes' Component Model*, Book chapter in *Builiding Reliable Component-Based Systems*, I. Crnkovic, M. Larsson editors, Archtech House publishers, ISBN 1-58053-327-2, 2002, pp. 375-386

[25] P. Klint, R. Lämmel, C. Verhoef. Toward an engineering discipline for grammarware, ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 14, N. 3, Juillet 2005

[26] T.R. Dean and al., *Agile Parsing in TXL*, Journal of Automated Software Engineering 10,4 (October 2003), pp. 311-336.

[27] J.M. Favre, *A New Approach to Software Exploration : Back-packing with* G^{SEE}, CSMR 2002

[28] L. Moonen, *Generating Robust Parsers using Island Grammars*, WCRE 2001

[29] J. Bézivin, N. Ploquin. *Tooling the MDA framework: a new software maintenance and evolution scheme proposal*. Journal of Object-Oriented Programming, JOOP, 2001

[30] J. Bézivin, H. Brunelière, F. Jouault, I. Kurtev. *Model Engineering Support for Tool Interoperability*, WiSME 2005, http://planetmde.org/wisme-2005, Oct. 2005

[31] I. Kurtev, J. Bezivin, and M. Aksit, *Technological spaces: an initial appraisal*, In CoopIS, DOA'2002 Federated Conferences, Industrial track, Irvine, 2002

[32] F. DeRemer, H. Kron. *Programming-in-the-Large vs. Programming-in-the-Small*. IEEE Transactions on Software Engineering, Vol. 2, N. 2, Fev. 1976, pp. 80-86.

[33] I. Bull, J.M. Favre, *Visualization in the Context of Model Driven Engineering*, International Workshop on Model Driven Development of Advanced User Interfaces, MDDAUI @ MoDELS 2005, Montego Bay, Jamaica, Octobre 2005

[34] J.M. Favre, *Maintenance et Ré-ingénierie globale des logiciels*, Thèse de doctorat, Université de Grenoble, 1996

[35] A. Mendelzon, J. Sametinger. *Reverse Engineering by Querying and Visualization*. in Software - Concepts and Tools, Vol. 16, N. 4, 1995, pp. 170-182

[36] R. Kazman, S. J. Carrière, *View Extraction and View Fusion in Architectural Understanding*, International Conference on Software Reuse, ICSR 1998

[37] S. Ducasse, M. Lanza, S. Tichelaar, *Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems*, 2nd International Workshop on Constructing Software Engineering Tools, CoSET 2002

[38] M. Lanza, *CodeCrawler: Lessons Learned in Building a Software Visualization Tool*, 2003

Extraction des connaissances et cartographie avec la plateforme Obeo Agility et Acceleo

Mise en oeuvre des technologies de reverse engineering et de cartographie par une approche IDM

Obeo

Obeo
2 rue Robert Schuman
44400 REZE
02 51 13 51 42
info@obeo.fr

RÉSUMÉ. *La cartographie de systèmes hétérogènes et la refonte outillée sont des besoins réels de l'informatique moderne mais elles nécessitent la mise au point d'outils complexes. Les technologies d'analyse de code et de transformation vers des méta-modèles visuels spécifiques peuvent être combinées dans une approche IDM afin d'apporter une réponse flexible et efficace à ces problèmes.*

ABSTRACT. *Cartography of heterogeneous systems and software recasting are real needs for modern data processing but they require the development of complex tools. Source code analysers and model transformations toward specific visual meta-models can be combined in an MDA approach in order to bring a flexible and effective response to these problems.*

MOTS-CLÉS : *Rétro-modélisation, Analyseur de surface, PSM, EMF, Eclipse, Cartographie, IDM, DSM.*

KEYWORDS: *Reverse engineering, Shallow parsing, PSM, EMF, Eclipse, Cartography, MDA/MDE, DSM.*

1. Introduction

La complexité grandissante des systèmes d'information et la nécessite de les aligner avec le métier de l'entreprise engendrent de nouveaux besoins de maîtrise et de gestion de l'évolution non résolu par les techniques actuelles.

Obeo a développé une technologie innovante d'analyse de systèmes logiciels dédiée aux personnes qui veulent tirer profit de l'IDM. Cette technologie appelée Obeo Agility apporte une grande flexibilité tant par le fait qu'elle permet la création de cartographies spécifiques que par l'hétérogénéité des sources qu'il est possible d'analyser.

2. Méta-modèles de code et cartographie dirigée par les modèles

Le processus actuel d'analyse d'un langage est une tâche complexe réalisée par un spécialiste des techniques de compilation : examen du respect des règles lexicales, syntaxiques, et sémantiques. Si l'on souhaite obtenir une sortie de type modèle, la phase de mise au point est encore plus longue et fastidieuse, induisant des coûts et des délais de réalisation très élevés.

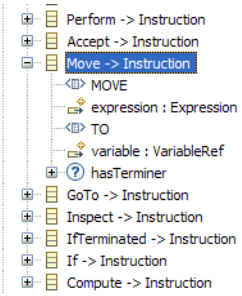
En combinant des technologies d'analyse de surface et de méta-modélisation, Obeo a conçu une démarche et une plateforme innovante basée sur eclipse pour extraire, visualiser et faire évoluer les applications existantes.

Les paragraphes suivants montrent un cas d'utilisation faisant intervenir cette démarche et l'outillage associé dans le cadre d'une analyse d'un programme à technologies hétérogènes Cobol et C qui, de plus, mélange interface graphique, logique métier et accès aux données.

1) Etape 1 : Mise au point des analyseurs

La mise en point d'une telle chaîne de reverse engineering commence par la création d'un méta-modèle de code qui est décoré avec des informations syntaxiques.

La création de ces méta-modèles de code grâce à Obeo Reverse permet de se reconcentrer sur le langage à analyser, car il fusionne les étapes d'analyse grammaticale, de méta-modélisation, et d'écriture des actions sémantiques permettant de produire les modèles.



Cette approche réduit considérablement le temps nécessaire à la réalisation d'un analyseur pour un langage donné. Cet analyseur dit "de surface" pourra ne récupérer que les informations macroscopiques (ici les appels aux programmes

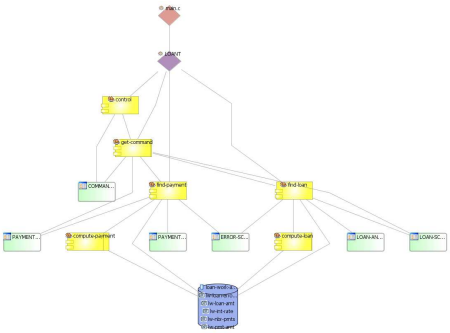
cobol depuis le programme C) ou détecter précisément tous les aspects du langages (on parle dans ce cas d'analyseur "fin").

2) *Etape 2 : Extraction d'une cartographie sur-mesure*

Les modèles produits par les analyseurs sont proches des technologies sources. Il reste donc à en extraire les informations pertinentes (diagramme d'architecture, modèle de haut niveau, règles métiers, ...).

Cela passe par la définition d'un méta-modèle de cartographie décrivant les artefacts à visualiser. Ce méta-modèle reflétera de manière immédiate la préoccupation que l'on cherche à combler par la visualisation.

Par la suite, une transformation de modèle permet simplement de peupler ce modèle de cartographie à partir du modèle issu de l'analyse de l'existant.

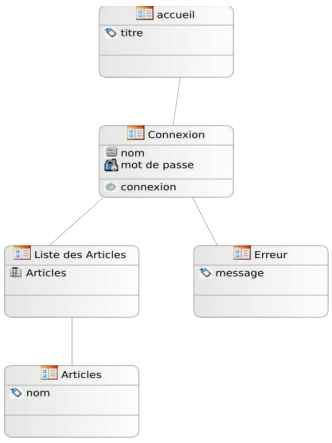


Enfin, la visualisation graphique du résultat se fait via un éditeur dédié au méta-modèle de cartographie (modeleur de type DSM).

Une des caractéristiques essentielles de l'ensemble de ce processus est l'aspect intégralement personnalisable des vues sur le modèle permettant de s'adapter à différents besoins.

3) *Etape 3 : Exploitation*

Grâce à l'approche IDM, les perspectives ouvertes deviennent très larges : en effet les cartographies obtenues sont autant de modèles exploitables en tant que tel pour de l'analyse d'impacts, des contrôles qualimétriques, du refactoring ou enfin pour les projets de refonte des applications informatiques.



Les principaux aspects novateurs de la plateforme Obeo Agility résident dans sa capacité à être adaptée et paramétrée pour tous types de technologies, ainsi que dans son aspect pragmatique qui en fait un outil prêt à être utilisé dans un environnement industriel.

Actes de l'atelier de travail
Motifs de méta-modélisation

Application de l'ingénierie dirigée par les modèles à l'ingénierie des langages

Pierre-Alain Muller, Didier Vojtisek, Zoé Drey, Olivier Barais

IRISA / INRIA Rennes
Rennes, France
{pierre-alain.muller, didier.vojtisek, zoe.drey, olivier.barais}@irisa.fr

Kermeta est un meta-langage orienté-objet, exécutable, destiné notamment à la simulation et à la transformation de modèles. Dans le cadre de cette démonstration, nous nous proposons de montrer comment Kermeta peut être appliqué dans le contexte de l'ingénierie des langages (typiquement spécifiques à un domaine).

Avec Kermeta, la modélisation d'un langage se décompose en trois étapes :

- l'expression de la syntaxe abstraite, sous la forme d'un meta-modèle,
- l'expression du lien entre la syntaxe abstraite et le domaine sémantique, sous la forme d'une description opérationnelle,
- l'expression de la syntaxe concrète, réalisée dans le cadre de cette démo par interfaçage avec le générateur d'éditeurs graphiques TopCaseD.

Un exemple basé sur des automates illustrera la démarche ; nous montrerons notamment :

- la construction de la partie statique du meta-modèle des automates au moyen d'outils disponibles dans la sphère Eclipse,
- le chargement de modèles (d'automates) conformes au meta-modèle,
- l'évaluation de contraintes statiques (invariants) sur ces modèles,
- la spécification de comportement dans les meta-modèles (écriture du corps des méthodes),
- l'exécution (simulation) des modèles, conformément au comportement défini précédemment,
- une application à la transformation de modèles.

Développement coordonné UML 2 / *Enterprise Java Beans*™

Franck Barbier

Université de Pau – PauWare Research Group
Avenue de l'université
BP 1155
64013 Pau CEDEX - France
Franck.Barbier@FranckBarbier.com

RÉSUMÉ. *L'objectif de ce tutoriel est de fournir une méthode rigoureuse de développement d'applications distribuées basées sur les EJBs. La description des types d'EJBs, de leurs propriétés canoniques et des contraintes de mise en œuvre, notamment le déploiement, mène à la fabrication de méta-modèles de type EJB/PSM. Les modèles métier UML 2/PIM sont alors dérivés vers des modèles techniques d'implémentation conformes aux méta-modèles EJB/PSM. Des études de cas sont abordées (BMP, CMP, Stateful/Stateless et MDB) avec accès complet au code et possibilité de téléchargement sur Internet.*

ABSTRACT. *The tutorial's goal amounts to providing a rigorous development method dedicated to the building of EJB-based distributed applications. EJBs' description including the canonical types of EJBs, their recognized properties and associated use constraints (deployment especially) lead to defining PSM-like metamodels. Business UML 2 (a.k.a. PIM) models are therefore transformed into technical implementation models which conform to the PSM-like metamodels. Some case studies are discussed (BMP, CMP, Stateful/Stateless and MDB) with complete code access and possible downloads from the Web.*

MOTS-CLÉS : *UML, composants logiciels, Enterprise Java Beans, ingénierie des modèles.*

KEYWORDS: *UML, software components, Enterprise Java Beans, Model-Driven Engineering.*

Actes de l'atelier de travail
Motifs de méta-modélisation

Adaptation dynamique d'assemblages de dispositifs dirigée par des modèles

Daniel Cheung-Foo-Wo[*,**] — **Mireille Blay-Fornarino**[*]
Jean-Yves Tigli[*] — **Stéphane Lavirotte**[*,***] — **Michel Riveill**[*]

[*] *Laboratoire I3S (CNRS - UNSA), 930 Route des Colles, 06 903 Sophia Antipolis*
{cheung,blay,tigli,lavirott,riveill}@polytech.unice.fr

[**] *CSTB, 290 Route des Lucioles, BP 209, 06 904 Sophia Antipolis*

[***] *IUFM, C. Freinet - Académie de Nice, 89 Av. George V, 06 046 Nice Cedex*

ABSTRACT. *Adaptable systems have been required since the emergence of pervasive computing. Adaptations are seen as direct consequences of the dynamic variations of devices surrounding the system. They consist in an integration cycle that is threefold : the discovery of new devices, the selection and the validation of adaptations. MDI allows to model the management of the variations of the devices availability. We use different representations of the same system at runtime according to different points of view. Transformations are applied at the metamodel level and maintain the coherence of the system. The metamodel and its tools are called "designer". We detail two designers and discuss the accuracy of our approach.*

RÉSUMÉ. *L'informatique ambiante augmente la demande en systèmes adaptables à des situations non prévues à l'avance. L'adaptation est la conséquence de l'apparition et la disparition dynamique de périphériques environnants, notés "dispositifs" dans l'article. Elle constitue un cycle qu'on nomme "cycle d'intégration" dont les phases sont : la découverte de ces périphériques, le choix puis la validation des adaptations à effectuer. L'IDM offre des moyens rigoureux pour exprimer cette découverte dynamique. En effet, nous proposons de représenter suivant différents modèles un même système pendant son l'exécution. Chaque représentation est définie par un métamodèle au niveau duquel sont développés des outils de manipulation. Les transformations entre les métamodèles conservent la cohérence entre les modèles. Le but de cet article est finalement de montrer comment exprimer les cycles d'intégration en utilisant les modèles. Nous détaillons deux modèles ISL et ADL (Wcomp) et les transformations associées.*

KEYWORDS: *Dynamic Adaptation, Model, Component.*

MOTS-CLÉS : *Adaptation Dynamique, Modèle, Composant.*

1. Introduction

Nous nous intéressons à l'adaptation de systèmes informatiques à des utilisations non prévues à l'avance par programmation. On définit le fonctionnement d'un système comme étant la combinaison de plusieurs assemblages de composants qui définissent chacun un aspect précis d'une application. Il existe plusieurs approches pour adapter un logiciel en combinant des assemblages. Ces approches ont été répertoriées sommairement dans (McKinley *et al.*, 2004a) et puis exhaustivement dans (McKinley *et al.*, 2004b).

Adaptation par composition

Une des méthodologies d'adaptation consiste à adapter en ré-assemblant des composants logiciels. C'est ce que McKinley appelle la *composition* ou dans notre cas, re-*composition*. Mais la définition de *composition* a beaucoup dérivé depuis plusieurs années. *Composer* pour McKinley consiste à regrouper des fonctionnalités décrites en s'appuyant sur un vocabulaire propre à un ou plusieurs domaines techniques. Et c'est sur cette définition théorique que nous nous appuyons pour réaliser nos adaptations. Nous posons le vocabulaire propre à notre application en choisissant de construire un modèle pour chaque domaine technique et chaque vue de l'application. Nous décrivons un métamodèle en UML pour chaque domaine. Finalement, nous utilisons un formalisme logique pour décrire les transformations entre les modèles.

Adaptation en fonction du contexte opérationnel

L'ensemble des dispositifs (périphériques informatiques) à un instant donné constitue le *contexte opérationnel* de l'application. C'est le contexte dans lequel le système interagit avec son environnement. L'intégration dynamique de dispositifs consiste dans un premier temps à détecter leur apparition et leur disparition. Puis, il s'agit de sélectionner les adaptations à effectuer selon le contexte opérationnel courant pour finalement mettre en oeuvre ces adaptations.

Selon ces différentes étapes, certains formalismes tels que les ADLs (Architecture Description Languages) ou les *règles de réécriture* se révèlent plus adaptés. Les ADLs décrivent les architectures logicielles. Les règles de réécriture capturent et regroupent les adaptations sous forme d'ensembles de règles (schémas) et facilitent leur sélection.

L'adaptation d'assemblages de composants exige dans notre cas l'utilisation de plusieurs métamodèles correspondant chacun à un domaine technique. Nous manipulons les modèles en utilisant soit des représentations graphiques de leurs éléments de base (c'est-à-dire leur vocabulaire), soit leur en associant une syntaxe concrète à travers un langage de description. Lorsque nous développons à partir de ces représentations graphiques ou lorsque nous décrivons des fonctionnalités par l'intermédiaire

du langage dédié, nous utilisons des éditeurs que nous appelons "designers" (en français, "concepteurs") adaptés pour chaque domaine.

Cet article montre d'abord comment exprimer l'intégration d'un nouveau *dispositif* en utilisant les *modèles*, puis comment intégrer les transformations de modèles pour garder l'application cohérente lors de son exécution. Nous décrivons deux exemples de designers qui sont ISL4Wcomp et Wcomp.

Wcomp, le designer ADL, permet l'instanciation, la destruction de composants logiciels et la gestion des liaisons entre composants. Ce designer est plus proche de l'assemblage de composants et facilite la compréhension et la modification de ces assemblages en termes d'ajout et de retrait de composants et de liaisons. On y définit les notions d'instanciation (add_{ADL}), de destruction ($\text{remove}_{\text{ADL}}$) de composants, d'ajout (bind) et de retrait (unbind) de liaisons.

ISL4Wcomp, le designer ISL, permet d'éditer des schémas décrivant des *interactions* (Blay-Fornarino *et al.*, 2004) entre des composants logiciels. Ce designer permet une description déclarative et facilite par conséquent l'expression des adaptations. Il permet également leur validation a priori. Y sont définies les notions d'ajout (add_{ISL}) et de retrait ($\text{remove}_{\text{ISL}}$) de schémas.

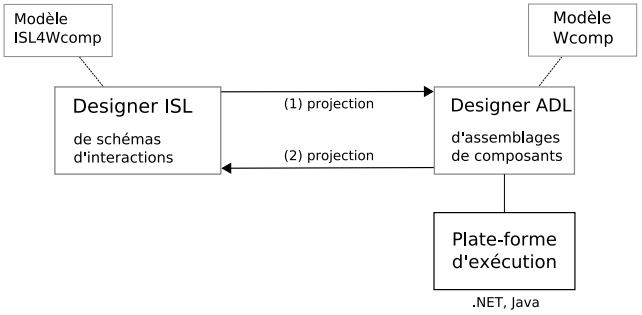


Figure 1. *Problématique provenant de nos applications*

Le but de ce travail est de gérer la cohérence entre différentes représentations d'un même système (cf. figure 1).

2. Vues différentes d'une application

Nous voyons dans cette section deux vues possibles d'une même application. Dans le domaine des systèmes multi-dispositifs, la vue la plus adaptée est celle décrite en ADL. D'une part, les systèmes *multi-dispositifs* sont des systèmes informatiques qui sont connectés à un très grand nombre de dispositifs. Ces systèmes n'interagissent avec leur environnement qu'à travers ces derniers. Notre expérience montre que le développement d'applications pour les systèmes multi-dispositifs se conçoit aisément en

s'appuyant sur les concepts des ADLs. D'autre part, le fait de représenter un logiciel comme un ensemble de *boîtes noires* (voir (Broy, 1996)) connaît une correspondance immédiate avec la structure des langages vers lesquels on projette tels que l'Assembleur, le C, le C++, le Java, le C# ou l'Objective-C. Dans ces langages, on peut faire correspondre la notion de boîte noire – ayant des entrées et des sorties – à une fonction (ou une procédure ou une méthode). La projection d'une application conçue à partir d'une description en ADL sur diverses plates-formes est par conséquent aisée.

2.1. *Wcomp*

Le premier designer que nous proposons s'appuie donc sur les ADLs. On l'appelle Wcomp. Ce designer permet de manipuler les éléments de première classe (principalement, les composants et les connecteurs) du paradigme des composants logiciels dans lequel une application multi-dispositifs est écrite.

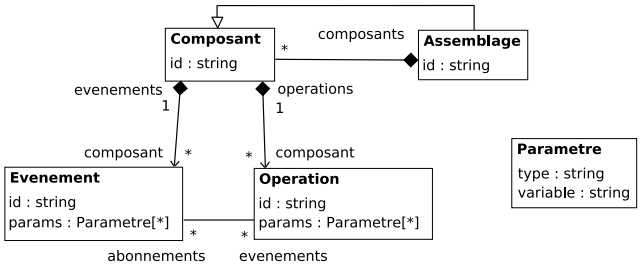


Figure 2. *Métamodèle Wcomp en UML (ADL)*

Dans la figure 2, nous avons établi le métamodèle de Wcomp. Une application est par définition un assemblage de composants. Cet assemblage peut être vu comme étant un nouveau composant. A l'image d'un composant matériel (par exemple, les circuits intégrés (McIlroy, 1968)), un composant logiciel est doté de ports d'entrée et de sortie qui sont respectivement implémentés par des évènements et des opérations. Les ports de sortie (les évènements) peuvent éventuellement être associés à certains ports d'entrée (les opérations).

2.2. *ISL4Wcomp*

Le designer ISL (ISL4Wcomp) s'appuie sur le langage de spécification d'interactions nommé ISL pour *Interaction Specification Language* (Berger, 2001, Blay-Fornarino *et al.*, 2004). ISL est un langage de description de schémas d'interactions entre des objets. Ce langage a la propriété d'être "composable" automatiquement. Cela signifie que l'on peut écrire plusieurs schémas indépendamment en ISL. Puis une fois ajoutés, ces schémas se composent pour ne donner plus qu'un seul schéma rassemblant *correctement* les fonctionnalités décrites séparément.

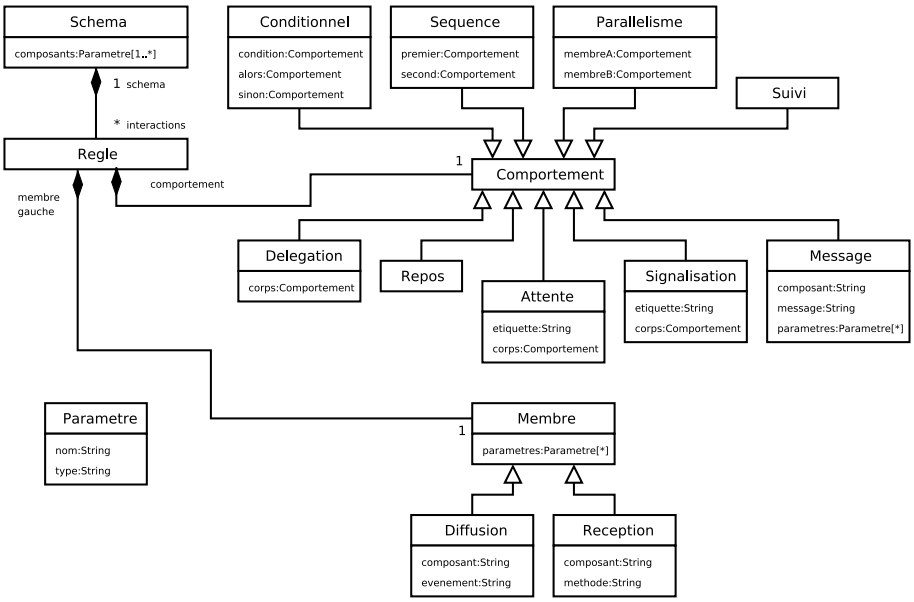


Figure 3. *Métamodèle ISL4Wcomp en UML (ISL)*

Afin d'utiliser les propriétés de ce langage dans nos expérimentations et dans notre formalisme, nous avons adapté ses opérateurs pour qu'ils prennent en charge la notion d'événement (Cheung-Foo-Wo *et al.,* 2005). Un schéma écrit en ISL4Wcomp est constitué d'un ensemble de règles de réécriture agissant sur des appels d'opérations ou de déclenchement d'événements. La grammaire du langage ISL pour le logiciel ISL4Wcomp et l'implémentation de la fonction de composition de schémas s'appuient sur les concepts de SRP (Système de Résolution de Problèmes).

Prolog étant actuellement le langage le plus connu pour un SRP, c'est dans ce langage que nous avons implémenté les règles logiques de la composition. L'implémentation consiste en 25 règles de composition. Ceci nous a permis en outre de vérifier certaines propriétés essentielles comme la commutativité de la composition des schémas. Au niveau d'ISL4Wcomp, nous ne considérons que quelques opérateurs. Ces opérateurs regroupent les notions d'appel d'opérations, de déclenchement d'événements, de séquence entre deux opérations, de concurrence, d'attente, de notification (ou signalisation) et de condition. Donc, nous avons neuf opérateurs :

$\alpha;\beta$ la séquence, α s'exécute après β

$\alpha \| \beta$ la concurrence, α et β s'exécutent en parallèle

if$(\gamma)\{\alpha\}$else$\{\beta\}$ la condition, α s'exécute si γ est vraie, sinon β s'exécute

$c.m()$ l'envoi d'un message, appel de la méthode m sur le composant c

$call$ se réfère à l'opération ou l'événement du membre gauche.

nop No OPeration, n'effectue rien.

delegate $\{\alpha\}$ la délégation ; l'appel de méthode ou l'envoi d'événement spécifié par le membre gauche de la règle est remplacé par α

wait(x, α) l'attente ; l'exécution de α ne se fera qu'une fois que la variable x sera signalée (par l'opérateur de signalisation)

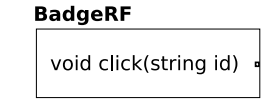
signal(x, α) la signalisation ; signale la variable x lorsque α a achevé son exécution

On note f_n la fonction de composition. Notons $s_1, \ldots, s_n$ des schémas d'interactions, pour tout n, $f_n(s_1, \ldots, s_n)$ est invariante par permutation de ces n schémas. Cela signifie en pratique que l'état dans lequel se trouve le système à un instant donné est indépendant de l'ordre dans lequel nous avons ajouté les schémas d'interactions.

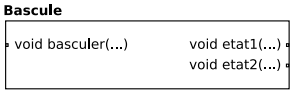
2.3. *Exemple d'utilisation*

Nous proposons l'exemple d'une application de commande à distance. Nous considérons sept composants logiciels qui sont regroupés dans la liste ci-dessous. Nous y trouvons le brochage du composant logiciel indiquant la signature de ses opérations (à gauche) et de ses événements (à droite) s'il en est doté, ainsi qu'une description intuitive de son comportement.

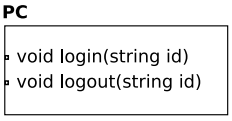
Voici un descriptif de la fonction de chaque composant :



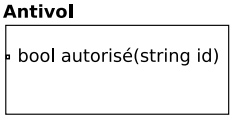
Le composant *BadgeRF* déclenche un événement *click* avec un identifiant unique *id* lorsque l'utilisateur appuie sur le bouton. Un BadgeRF (dispositif) est constitué d'un bouton et d'un émetteur.



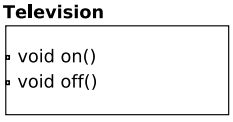
Le composant *Bascule* déclenche alternativement les événements *etat1* et *etat2* après l'appel de l'opération *basculer*. Cet opération accepte tout type de paramètres qu'il retransmet aux événements en sortie.



Le composant *PC* permet de se connecter à un PC de bureau en ouvrant son compte personnel. L'identification est effectuée par l'identifiant *id*.

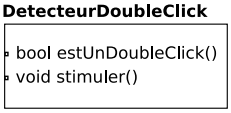


Le composant *Antivol* (authentification de l'utilisateur) vérifie que l'identifiant unique *id* en paramètre de l'opération *autorisé* est valide. Si tel est le cas, alors l'opération renvoie le booléen vrai, sinon elle renvoie faux.



Le composant *Television* permet de contrôler l'allumage et l'extinction d'une télé en appelant respectivement l'opération *on* ou *off*.

Le composant *Selecteur* permet de filtrer les paramètres d'entrées noté p_0 de l'opération *in* et de déclencher l'événement *out* ayant comme paramètres p_1 qui est une liste incluse dans p_0. Le filtrage est paramétré lors de l'instanciation du Selecteur.



Le composant *DetecteurDoubleClick* mémorise l'instant où il a été stimulé par l'appel de l'opération *stimuler.* Si entre deux stimulations, il s'est déroulé plus de $\delta = 500$ms par exemple, le prochain appel de l'opération *estUnDoubleClick* renverra le booléen faux. Sinon, elle renverra le booléen vrai.

L'application sur laquelle nous nous appuyons est schématisée dans la figure 4. Elle s'exécute sur un PC de bureau. Le composant BadgeRF est un capteur accessible via un réseau sans-fil et les autres composants sont liés localement au PC.

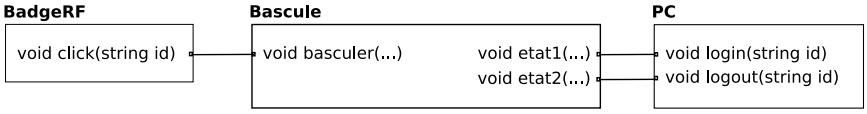


Figure 4. *Assemblage représentant l'application "télécommande".*

Scénario :

Cette application consiste à se connecter sur une machine de type PC à l'aide d'un badge RF (Radio Frequency). Le scénario est tel que lorsque l'on clique sur le bouton du badge RF, le composant BadgeRF déclenche l'événement *click* avec comme paramètre, l'identifiant de l'utilisateur. L'opération *basculer* est abonnée à ce dernier événement (voir la figure 4). Ainsi, nous avons : soit l'évènement *etat1* est déclenché, ce qui a pour effet d'appeler l'opération login sur le PC avec l'identifiant ; soit l'évènement *etat2* est déclenché, ce qui a pour effet d'appeler plutôt l'opération *logout* sur le PC. Dans le premier cas, la session appartenant à l'utilisateur est ouverte et dans le second, elle est refermée.

Adaptation :

Concernant l'adaptation de cette application, nous avons deux possibilités : soit nous travaillons dans le modèle Wcomp et instancions des composants ou introduisons des liaisons, soit nous basculons vers le modèle ISL pour décrire et ajouter des schémas d'interactions.

Nous proposons l'écriture de deux exemples de schémas d'interactions. Dans un premier schéma, nous voulons intégrer un composant supplémentaire dans l'application qui est "l'*antivol*" (ou l'authentification de l'utilisateur). Dans un second schéma, nous ajoutons un dispositif supplémentaire qui est une télévision accompagnée d'un composant logiciel sélecteur de paramètres et d'un autre qui détecte les doubles clicks d'un bouton.

Le premier schéma d'interactions est décrit dans le tableau 1.

n°	Nom du schéma	Description
1	schéma_antivol	**Au lieu de** se connecter au PC, **si** l'antivol l'autorise, **alors** "**faire ce qui est prévu**" (c'est-à-dire, se connecter au PC), **sinon** "**ne rien faire**".

Table 1. *Schéma de l'antivol*

Puisque certains composants logiciels sont associés à des dispositifs, nous pouvons utiliser ISL4Wcomp pour les intégrer simplement en ajoutant des schémas d'interactions. Le second schéma que nous mettons en oeuvre exprime le comportement décrit dans le tableau 2.

n°	Nom du schéma	Description
2	schéma_tv	**Au lieu de** se connecter au PC, **si** le détecteur de double clicks en a détecté un, **alors** "**faire ce qui est prévu**" (c'est-à-dire se connecter au PC), **sinon** allumer la télévision.

Table 2. *Schéma d'intégration de la télévision*

Exemple de schémas

Nous reprenons la première description intuitive du schéma de "l'antivol de session" (voir tableau 1). Sa traduction en ISL, une fois appliquée aux composants *pc* et *antivol*, est la suivante :

```
schema schéma_antivol (pc,antivol) {
  pc.login(int id) {
    if(antivol.autorise(int id)) {call} else {delegate{nop}}
  }
}
```

Figure 5. *Schéma n°1 : l'antivol (ou authentification de l'utilisateur).*

Nous reprenons également la seconde description du schéma "d'intégration de la télévision" (voir tableau 2). Sa traduction en ISL4Wcomp une fois appliquée aux composants *pc*, *télévision* et *détecteur* est résumée par la figure 6.

Transformation T_{ISL} : composition de schémas

Le composition de schémas d'interactions se fait en plusieurs phases. On part de l'hypothèse que l'on compose des schémas quelconques.

```
schema schéma_tv (pc,television,detecteur) {
  pc.login(int id) {
    if(detecteur.unSeulClick()) {call} else {delegate{television.on()}}
  }
}
```

Figure 6. *Schéma n°2 : l'intégration de la télévision.*

1) Nous devons unifier les paramètres. Soient $p_1, \ldots, p_n$ les paramètres de ces n schémas. Pour créer un unique schéma ayant des paramètres p, on unifie tous les paramètres c'est-à-dire qu'on crée une liste de paramètres qui est l'union de tous les p_i.

2) Puis, on sépare les règles des schémas afin d'avoir les règles comme des entités de première classe et ne travailler que sur celles-ci. On se retrouve donc avec un ensemble de règles et une liste de paramètres unifiée p.

3) On groupe les règles ayant le même membre gauche (qu'il soit une réception ou une diffusion). Lorsqu'un groupe est constitué d'une seule règle, alors cette règle figure sans modification dans le schéma résultant. Lorsqu'un groupe est constitué de plusieurs règles, alors les corps de ces règles doivent fusionner. Le résultat de cette fusion est un nouveau comportement. L'ensemble est constitué du membre gauche et de ce nouveau comportement figure dans le schéma résultant.

4) La fusion est une fonction n-aire qui prend n comportements comme paramètres d'entrée. Elle est définie à partir d'une fonction de fusion binaire commutative. Par extension, nous pourrions considérer que la fonction de fusion est une transformation du même modèle. Toutefois, nous avons pour cet article développé les transformations entre les modèles Wcomp et ISL4Wcomp.

```
schema schéma_courant (pc,television,detecteur,antivol) {
 pc.login(int id) {
  if(antivol.autorise(id)) {
   if(compteur.unSeulClick()) { call }
   else { delegate { television.on() } }
  } else {
   if(compteur.unSeulClick()) {delegate {nop}}
   else {delegate {television.on()}
  }
 }
}
```

Figure 7. *Schéma résultant de la composition des schémas n°1 et n°2.*

Le résultat de la composition des schémas d'interactions a la signification suivante : lorsque l'opération *pc.login* est appelée, au lieu d'appeler directement l'opération sur le composant pc, on vérifie conditions suivantes. Si l'antivol nous l'autorise et qu'on a effectué qu'un click, alors on appelle effectivement *pc.login*. Si en revanche,

on a effectué deux clicks, alors on allume la télévision. Si l'antivol ne donne pas son approbation et qu'on a effectué qu'un click, alors rien ne se passe. Si en revanche, on a effectué deux clicks, alors on allume la télévision. Quelque soit la réponse de l'antivol, l'allumage de la télévision reste indépendant.

3. Cycle d'intégration d'un nouveau dispositif

Dans le paradigme orienté-composant, un composant est doté de ports. Une *liaison* représente l'abonnement d'un évènement qu'un composant peut diffuser (port de sortie) à un appel d'une *opération* (port d'entrée). Un évènement s'accompagne souvent de données sous forme de paramètres. Un *schéma d'interactions* ajoute des *composants de contrôle* entre les ports d'entrée et de sortie. Ces composants de contrôle se connectent soit avant la *réception* de certains messages, soit après la *diffusion* d'évènements. Ces schémas sont exprimés sous forme de règles de réécriture en ISL.

Dans un assemblage, certains composants représentent des dispositifs tels que des capteurs et des actionneurs. L'adaptation d'un assemblage se fait de façon automatique. Elle s'appuie sur une boucle de découverte de dispositifs. La découverte d'un nouveau dispositif ou sa disparition (lors d'une panne, une déconnexion) implique la modification de l'application. Cette modification se passe pendant son exécution. Elle consiste en deux étapes :

1) en instanciant ou détruisant des composants logiciels

2) en sélectionnant et en ajoutant des schémas d'interactions

Scruter l'ensemble des dispositifs et réagir à des modifications éventuelles de cet ensemble est un mécanisme itératif qu'on appelle *cycle d'intégration d'un dispositif*. D'autres adaptations sont possibles en réaction à la découverte d'un dispositif. Citons par exemple les adaptations manuelles, la gestion de stratégies (Buisson *et al.*, 2005) et les adaptations par aspect (Barais *et al.*, 2006).

Designers

Au niveau d'ISL4Wcomp, l'ajout d'un schéma d'interactions a pour conséquence la construction d'un assemblage de composants. Comment se déroule cette construction ? Un algorithme permet de valider cet assemblage et de construire le graphe des interactions en conséquence. Si celui-ci est cohérent, il est projeté vers le Wcomp (étape 1, figure 1). Il a pour conséquence l'ajout de liaisons et la création de *composants de contrôle*. Un composant de contrôle est un composant qui a une sémantique spécifique et connu dans le graphe de composants. Si le graphe des interactions est incohérent, aucune projection n'est entreprise.

Au niveau du designer Wcomp, l'adaptation d'un assemblage de composants consiste à instancier ou détruire des composants logiciels afin d'intégrer des dispositifs. Elle consiste également à introduire ou rompre des liaisons entre des composants.

Cohérence

Ajout d'une liaison. Afin de maintenir la cohérence entre les deux designers, chaque introduction de liaison est transformée en un schéma d'interactions. Ce schéma est ensuite ajouté dans ISL4Wcomp (étape 2.1, figure 1).

Retrait d'une liaison. La rupture d'une liaison implique le retrait du schéma d'interactions correspondant dans le designer ISL4Wcomp.

Si la transformation due à l'introduction de liaisons génère une incohérence, alors les deux designers deviennent temporairement incohérents. Pour y remédier, l'intégralité de l'assemblage de composants dans Wcomp est d'abord transformée en un schéma unique dans ISL4Wcomp (étape 2.2, figure 1). Ce schéma sera le seul qui est présent à cet instant dans ISL4Wcomp. Cela signifie qu'on perd l'historique des schémas d'interactions précédemment ajoutés.

Nous avons décrit la manière dont ces deux designers collaborent selon deux points de vue d'une même application :

– les schémas et leur composition

– les composants et leurs liaisons

Nous voyons en détail dans la section suivante les deux transformations de modèles. Chacun a comme but de s'adresser à un domaine technique différent : la description d'architectures logicielles et la séparation de préoccupations.

4. Transformations de modèles

Nous analysons les modifications possibles dans chaque modèle. Nous allons voir deux transformations entre Wcomp et ISL4Wcomp. Nous distinguons les transformations totales d'un modèle à un autre et les transformations partielles.

Transformations totales. Ces transformations peuvent être des *transformations totales* d'un modèle à un autre. La totalité du modèle est transformée dans un autre modèle. Nous les appelons des *projections*. On projette un modèle sur un autre modèle.

Transformations partielles. Ces transformations peuvent être des *transformations partielles*, c'est-à-dire qu'elles ne concernent qu'un sous-ensemble des modifications. Nous les appelons des *perturbations*. Une partie du modèle est modifiée. Uniquement cette partie *perturbée* est transformée dans un nouveau modèle.

Nous proposons d'abord l'étude des modifications élémentaires dans le modèle Wcomp. Il s'agit de l'ajout, du retrait de liaisons. On identifie alors les opérations élémentaires suivantes : add_{ADL}(id, type), $\text{remove}_{\text{ADL}}$(id), bind(id, id_{source}, id de l'évènement,id_{cible}, id de l'opération), unbind(id). Dans ISL4Wcomp, les modifications élémentaires se résument aux actions suivantes : on peut poser ou retirer

un schéma d'interactions. On identifie alors les opérations élémentaires suivantes : add_{ISL}(id du schéma, schéma), $\text{remove}_{\text{ISL}}$(id du schéma).

4.1. *Transformation* $T_{ADL \rightarrow ISL}$

C'est l'étape 2 de la figure 1 de la page 77. On suppose que le modèle ISL se trouve dans un état noté E_0^{ISL}. Wcomp effectue une modification notée δ_w. Quelle est la modification à apporter au modèle ISL pour intégrer cette modification δ_w et se trouver dans un état noté E_1^{ISL} ?

$$E_0^{\text{ISL}} + T_{\text{ADL}\rightarrow\text{ISL}}(\delta_w) = E_1^{\text{ISL}}$$

La transformation que nous étudions se note $T_{\text{ADL}\rightarrow\text{ISL}}$. Cette transformation peut être séparée en deux sous-transformations, chacune travaillant sur un sous-ensemble du domaine des modifications de Wcomp. On les note :

$$T^l_{\text{ADL}\rightarrow\text{ISL}}(w) \text{ et } T^r_{\text{ADL}\rightarrow\text{ISL}}(w).$$

L'exposant l signifie que cette sous-transformation va prendre en compte uniquement les *liaisons* introduites entre deux ports. L'exposant r signifie que cette sous-transformation va prendre en compte les *ruptures* de liaisons du modèle.

Introduction d'une liaison : $T^l_{ADL\rightarrow ISL}(w)$

Le premier cas consiste à lier une opération o à un événement e. On note respectivement c_1 et c_2 les composants associées à ces deux pattes. On a δ_w= bind (id, c_1, e,c_2, o). La règle de transformation logique $T^l_{\text{WI}}(\delta_w)$ se note (similairement au formalisme Typol (Despeyroux, 1988)) :

$$\frac{}{\vdash bind(id, c_1, e, c_2, o) : add_{ISL}(id, s)}\,(T^l_{\text{WI}})$$ où s est le schéma suivant :

```
schema s (c1,c2) {
  c1.e(params) {
    call || c2.o(params)
  }
}
```

Rupture de liaison : $T^r_{ADL\rightarrow ISL}(w)$

Le second cas consiste à rompre la liaison entre une opération o à un événement e. On note respectivement c_1 et c_2 les composants associées à ces deux pattes. On a δ_w= unbind (id). La règle de transformation logique $T^r_{\text{WI}}(\delta_w)$ se note :

$$\frac{}{\vdash unbind(id) : remove_{ISL}(id)}\,(T^r_{\text{WI}})$$ où id est l'identifiant d'un schéma existant.

Cas d'échec

Si le schéma précédent n'existe pas (c'est-à-dire, si $T^r_{\text{ADL}\rightarrow\text{ISL}}$ n'est pas valide), alors l'assemblage de composants dans Wcomp est transformé dans son intégralité en un schéma d'interactions. Nous expliquons l'algorithme de transformation (étape 2, figure 1, page 77) en se basant sur les métamodèles.

Algorithme de tranformation

On classe d'abord les composants Wcomp en deux catégories : ceux qui sont spécifiques à ISL4Wcomp (comme par exemple, le composant *Selecteur*) et ceux qui ne le sont pas comme les composants représentant les dispositifs et les autres composants logiciels. On note A l'ensemble des composants spécifiques. On distingue les liaisons dont les extrémités restent dans l'ensemble A des autres liaisons (voir figure 8). On les groupe dans un ensemble noté A_l. L'ensemble des composants de A liés par les liaisons de A_l constitue une forêt dont les noeuds sont les composants. Chaque arbre constitue une règle en ISL4Wcomp.

On considère deux cas : le cas où il existe une unique liaison entre un composant "autre" et un composant spécifique et le cas où il en existe plusieurs.

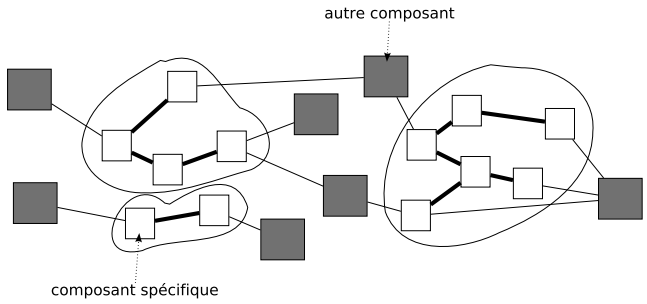


Figure 8. *Forêt de composants spécifiques.*

Dans le premier cas, on note l l'unique liaison entre un composant noté "*autre*" (c'est-à-dire non spécifique) et un composant spécifique. Le membre gauche de la règle est l'évènement associé à la liaison (voir un exemple de graphe de composants et de tri dans la figure 8). L'arbre de composants spécifiques est un "arbre ISL" selon (Berger, 2001). On dispose de toutes les informations nécessaires pour construire la règle.

Dans le second cas, on note $l_1, \ldots, l_n$ ces liaisons. On oriente le graphe dans le sens de l'évènement vers l'opération. Chaque arbre partant de l'événement associé à l_i représente alors le comportement d'une règle. Nous avons, dans ce cas, n règles. Toutes ces règles sont ensuite regroupées dans un unique schéma d'interactions.

L'apparition de cycles dans un arbre génère une erreur de transformation car cela n'a pas de sens en ISL4Wcomp.

4.2. *Transformation* $T_{ISL \rightarrow ADL}$

Dans le designer ISL4Wcomp, un schéma se compose de plusieurs règles. Chaque règle contient un unique corps qui explicite la réécriture du membre gauche. Ce corps est constitué d'un ensemble d'opérateurs imbriqués. Le corps d'une règle peut alors être représenté par un arbre dont les noeuds sont les opérateurs (voir l'exemple de la figure 9 et 10).

Réécriture d'un appel de méthode

Nous avons deux cas. Si le membre gauche (indiquant un composant c, une opération o) est une réception, alors le composant c se retrouve après le $call$ dans l'arbre ISL (voir figure 9). Ce sont en effet les interactions qui portent sur son opération o qui sont redéfinies. On note $M_g[c, m]$ le membre gauche d'une règle d'interactions. La variable c représente le composant sur lequel agit la réception (c'est-à-dire l'appel de méthode) ou la diffusion d'évènements ayant comme identifiant m. Le symbole $P[c, m]$ représente l'évènement du noeud (c'est-à-dire le composant) précédent. Soit la règle "call" :

$$\frac{\vdash M_g \text{ est une réception}}{M_g[c_0, m_0], P[c, m] \vdash call : bind(id, c, m, c_0, m_0)} \text{ (call) où } id \text{ est un identifiant unique.}$$

La règle précédente se lit de la manière suivante : si le membre gauche est une réception, alors on évalue $call$ et le résultat de cette évaluation consiste à effectuer la modification $bind(id, c, m, c_0, m_0)$ qui signifie que l'opération m_0 du composant c_0 pointés par le membre gauche est reliée au dernier évènement caractérisé par le composant c et la méthode m. L'identifiant id est généré à partir des quatre paramètres c, m, c_0 et m_0. Pour ne pas avoir de conflits, les liaisons ayant déjà cet identifiant sont renommées.

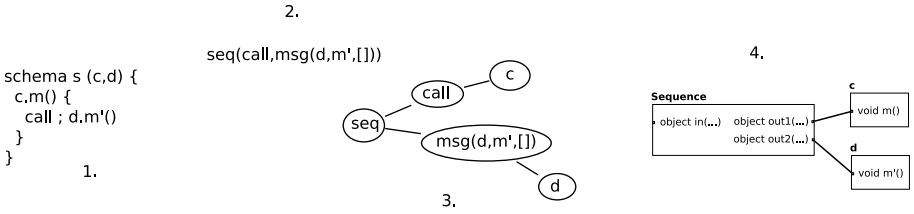


Figure 9. *Transformation d'une règle de réception.*

(La gestion des paramètres consiste en quelques règles logiques supplémentaires qui n'apparaîtrons pas dans cet article).

Pour ce qui est du problème du composant qui précède le premier noeud de l'arbre, sa valeur dépend du type du membre gauche. Si c'est une diffusion alors on établit une

liaison entre le composant indiqué par le membre gauche et la racine de l'interaction. Si c'est une réception, alors c'est une redirection qu'on réalise dans l'assemblage de composants. Il s'agit de propager la réécriture aux interactions des schémas résultants. Ainsi deux phases se révèlent alors nécessaires. D'abord, on doit travailler sur les diffusions et on crée un schéma partiel intermédiaire. Puis, on travaille sur les réceptions pour ajouter des composants de contrôle supplémentaires et compléter le schéma intermédiaire.

Diffusion d'un évènement

$$\frac{\vdash M_g \text{ est une diffusion}}{M_g \vdash call : \varnothing} \quad \text{(call)}$$

Si le membre gauche (indiquant un composant c) est une diffusion, alors le composant désigné c se retrouve à la racine de l'arbre ISL (voir figure 10). C'est l'effet de la diffusion qui est alors redéfinie.

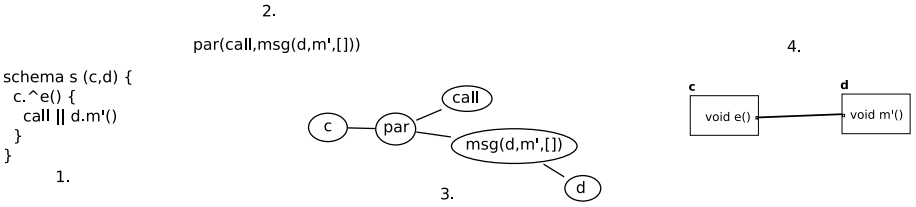


Figure 10. *Transformation d'une règle de diffusion.*

Projection dans Wcomp

La dernière phase consiste à faire correspondre l'arbre intermédiaire ISL à un assemblage de composants dans Wcomp.

Voici un exemple de règle de transformation concernant l'opérateur de concurrence. L'évaluation de l'opérateur de concurrence dans le contexte d'un membre gauche M_g et d'un parent P consiste à évaluer récursivement les deux branches α et β dans ce même contexte M_g,P.

$$\frac{\begin{array}{c} M_g, P \vdash \alpha : r_\alpha \\ M_g, P \vdash \beta : r_\beta \end{array}}{M_g, P \vdash par(\alpha, \beta) : \left\{ \begin{array}{l} r_\alpha \\ r_\beta \end{array} \right.} \quad \text{(concurrence)}$$

5. Discussion

Nous constatons que la multiplicité des métamodèles permet d'aborder l'interopérabilité entre plates-formes, mais suppose un enrichissement des plates-formes

au niveau "méta". En effet, la partie projection vers l'implémentation de Wcomp en .NET s'appuie actuellement sur une API de mise à jour des assemblages. De même, dans le cadre de notre application, nous avons besoin des remontées d'informations provenant des cibles (modification d'assemblage, ajout de schéma, découverte d'un nouveau dispositif) pour prendre en charge les modifications au niveau des modèles (Blay-Fornarino *et al.,* 2005). Ainsi l'interconnexion entre les modèles et leur mise en oeuvre est gérée par des mécanismes de réflexions sur les plates-formes cibles.

L'interopérabilité n'intervient pas de manière similaire en fonction des dépendances entre les modèles. En effet, la découverte de nouveaux dispositifs est basée sur un métamodèle non décrit dans cet article qui cible les mécanismes de découverte dynamique des infrastructures des réseaux et des middlewares sous-jacents. Celui-ci permet d'expliciter le procédé de mise à jour des adaptations indépendamment des plates-formes cibles. Néanmoins, la mise en oeuvre de Wcomp repose sur des *représentations* de ce métamodèle ; nous nous trouvons alors dans le cadre d'un empilement de modèles (Favre, 2004, Marvie *et al.,* 2006). En conséquence, celles-ci jouent le rôle de la plate-forme d'exécution pour les mises en oeuvre de Wcomp et les notifications de modifications qui sont aujourd'hui gérées directement entre ces implémentations sans passer par les modèles.

Dans une première approche, nous avons fait le choix de définir un métamodèle pivot à partir duquel serait gérée la cohérence de l'ensemble des designers via des transformations. Or, cette solution nous semble aujourd'hui peu efficace. En effet, nous avons vu qu'ISL4Wcomp gère la cohérence de certaines modifications ce qui implique qu'une modification qui est valide dans un métamodèle ne l'est plus dans un autre. Le modèle Wcomp est lui particulièrement bien adapté à définir des transformations efficaces vers des plates-formes d'implémentation. Par contre, l'analyse des graphes de propagation par transformation des graphes d'interactions est plus facile en partant du modèle ISL4Wcomp, tandis que des reconnaissances de *patterns* d'architectures tels que ceux présentés dans TranSat (Barais *et al.,* 2006) s'appliquera mieux à la représentation Wcomp. En conséquence dans notre cas d'étude, la composition des correspondances ou *mappings* (Pottinger *et al.,* 2003) pour simplifier la gestion de cohérence entre les différents designers nous semble mieux adaptée qu'une approche par métamodèle pivot. Une étude plus poussée devrait nous permettre de comparer ISL aux métamodèles définis dans le domaine des langages d'aspects (Ubayashi *et al.,* 2005, Han *et al.,* 2005).

6. Conclusion

L'ingénierie des modèles propose l'exploitation des modèles non seulement pour la production de code, mais également pour la génération de tests (Dubois *et al.,* 2005, Jézéquel *et al.,* 2006), la validation a priori des codes et des adaptations (Barais *et al.,* 2006) et la constitution de référentiels. Notre travail s'inscrit clairement dans cette démarche et aborde plus particulièrement la gestion de l'interopérabilité entre plates-formes via des métamodèles ainsi que des expérimentations de l'informatique

ambiante (Muñoz *et al.*, 2004). En l'occurrence, nos cibles sont aujourd'hui, pour la partie exécution des dispositifs, les middlewares usuels tels que .NET, Javaxxx. Pour la partie reconnaissance des adaptations à mettre en place, nous préférons actuellement une approche basée sur la logique des prédicats avec une mise en oeuvre en Prolog car les règles de projection s'y exprime de façon immédiate. Les transformations entre modèles sont définies au niveau des métamodèles et sont tout à fait indépendantes des cibles. Exprimées en Prolog, elles s'appuient sur des représentations exprimées sous la forme de faits : nous nous situons donc dans le cadre d'un même méta-métamodèle. D'autres solutions auraient probablement pu s'appliquer dont l'utilisation du MOF et les langages de transformation Kermeta (Fleurey *et al.*, 2006) ou ATL (Bézivin *et al.*, 2003).

7. References

Barais O., Lawall J., Meur A.-F. L., Duchien L., « Safe Integration of New Concerns in a Software Architecture », *13th Annual IEEE International Conference on Engineering of Computer Based Systems ECBS'06*, Potsdam, Germany, March, 2006.

Berger L., Mise en Oeuvre des Interactions en Environnements Distribués, Compilés et Fortement Typés : le Modèle MICADO, Thèse de doctorat, Université de Nice-Sophia Antipolis - Faculté des sciences et techniques, Ecole doctorale STIC - Informatique, Octobre, 2001.

Blay-Fornarino M., Charfi A., Emsellem D., Pinna-Dery A.-M., Riveill M., « Software interactions », *Journal Of Object Technology*, vol. 3, n° 10, p. 161-180, 2004.

Blay-Fornarino M., Franchi P., Nano O., « Vers une sémantique « plug-in » pour les modèles », 2005.

Broy M., « Towards a Mathematical Concept of a Component and Its Use », Components' Users Conference CUC'96, 1996.

Buisson J., André F., Pazat J.-L., « A framework for dynamic adaptation of parallel components », *ParCo 2005*, 2005. To appear.

Bézivin J., Dupé G., Jouault F., Pitette G., Rougui J., « First Experiments with the ATL Transformation Language : transforming XSLT into Xquery », *OOPSLA Workshop*, 2003.

Cheung-Foo-Wo D., Blay-Fornarino M., Tigli J.-Y., Dery A.-M., Emsellem D., Riveill M., « Langage d'aspect pour la composition dynamique de composants embarqués », *2ème Journée Francophone sur le Développement de Logiciels Par Aspects*, 2005.

Despeyroux T., TYPOL : a formalism to implement natural semantics, Technical report, INRIA - Sophia Antipolis, 1988.

Dubois H., Gérard S., Mraidha C., « Un langage d'action pour le développement UML de systèmes embarqués temps-réel », *IDM05, Actes des 1ères Journées sur l'Ingénierie Dirigée par les Modèles*, p. 175-192, 2005.

Favre J. M., « Towards a Basic Theory to Model Driven Engineering », *3rd Workshop in Software Model Engineering*, 2004.

Fleurey F., Drey Z., Didier V., *Kermeta language, Reference Manual*, IRISA. February, 2006.

Han Y., Kniesel G., Cremers A. B., « Towards Visual AspectJ by a Meta Model and Modeling Notation », *Proceedings of 6th International Workshop on Aspect-Oriented Modeling*, Held in conjunction with AOSD'05, Chicago, Illinois, USA, March, 2005.

Jézéquel J.-M., Gérard S., Mraidha C., Baudry B., *Le génie logiciel et l'IDM : une approche unificatrice par les modèles*, chapter 1, p. 1, 2006.

Marvie R., Duchien L., Blay-Fornarino M., *Les plates-formes d'exécution et l'IDM*, Edition Hermes, chapter 4, 2006.

McIlroy M. D., « Mass produced software components », *in* , B. R. E. P. Naur (ed.), *Software Engineering : Report on a Conference Sponsored by the NATO Science Committee*, 1968.

McKinley P. K., Sadjadi S. M., Kasten E. P., Cheng B. H. C., « Composing Adaptive Software », *IEEE Computer*, vol. 37, n° 7, p. 56-64, 2004a.

McKinley P. K., Sadjadi S. M., Kasten E. P., Cheng B. H. C., A Taxonomy of Compositional Adaptation, Technical Report n° MSU-CSE-04-17, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, May, 2004b.

Muñoz J., Pelechano V., Fons J., « Model Driven Development of Pervasive Systems », *International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, vol. 1, p. 3-14, june, 2004.

Pottinger R., Bernstein P. A., « Merging Models Based on Given Correspondences », *29th International Conference on the Very Large Data Bases*, 2003.

Ubayashi N., Moriyama G., Masuhara H., Tamai T., « A parameterized interpreter for modeling different AOP mechanisms », *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ACM Press, New York, NY, USA, p. 194-203, 2005.

ANNEXE POUR LE SERVICE FABRICATION

A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LES ACTES :
   IDM'06, Lille – Secondes Journées sur l'ingénierie des modèles

2. AUTEURS :
   Daniel Cheung-Foo-Wo[*,**] *— Mireille Blay-Fornarino*[*]
   Jean-Yves Tigli[*] *— Stéphane Lavirotte*[*,***] *— Michel Riveill*[*]

3. TITRE DE L'ARTICLE :
   Adaptation dynamique d'assemblages de dispositifs dirigée par des modèles

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   Adaptation dynamique d'assemblages

5. DATE DE CETTE VERSION :
   8 juin 2006

6. COORDONNÉES DES AUTEURS :

– adresse postale :

[*] Laboratoire I3S (CNRS - UNSA), 930 Route des Colles, 06 903 Sophia Antipolis

{cheung,blay,tigli,lavirott,riveill}@polytech.unice.fr

[**] CSTB, 290 Route des Lucioles, BP 209, 06 904 Sophia Antipolis

[***] IUFM, C. Freinet - Académie de Nice, 89 Av. George V, 06 046 Nice Cedex

– téléphone : 04 92 96 51 82

– télécopie : 04 92 96 50 55

– e-mail : cheung@polytech.unice.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes.cls`,
   version 1.2 du 03/03/2005.

8. FORMULAIRE DE COPYRIGHT :
   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com

La problématique de l'évolution structurelle dans les architectures logicielles à base de composants

Nassima Sadou, Dalila Tamzalit, Mourad Oussalah
Université de Nantes, Nantes Atlantique Universités, CNRS
LINA, FRE 2729, 2 rue de la Houssinière
BP 92208 44322 Nantes, F-44000 France
Tel : (33) 2-51-12-58-01/50/31
{Nassima.Sadou,Dalila.Tamzalit,Mourad.Oussalah}@univ-nantes.fr

Résumé. Nous abordons dans cet article la problématique de l'évolution structurelle dans les architectures logicielles à base de composants. Nous constatons que peu d'ADLs intègrent l'évolution dans leurs processus de développement. Pour contribuer à faire face à ce manque nous proposons SAEV, un modèle d'évolution pour les architectures logicielles. SAEV considère les éléments architecturaux comme des éléments de première classe, et ce sur trois niveaux d'abstraction : le niveau méta, le niveau architectural et le niveau application. SAEV considère et spécifie l'évolution sur ces différents niveaux d'abstraction de manière uniforme, ce qui représente une première de ses caractéristiques. L'exécution des opérations d'évolution est assurée au travers de la sélection et du déclenchement de règles d'évolution appropriées. Ces règles d'évolution, décrites selon le formalisme ECA représentent la seconde caractéristique de SAEV. Ces règles doivent respecter les invariants associés aux éléments architecturaux pour garantir la cohérence de l'architecture. L'ensemble des règles décrivant l'évolution d'un élément architectural est regroupé dans une stratégie d'évolution qui est associée à ce dernier. En considérant les différents niveaux d'abstraction, SAEV vise à être un modèle générique, uniforme et ouvert, permettant ainsi de décrire l'évolution à un niveau d'abstraction élevé et faciliter ainsi sa réutilisation et son extension.

Abstract. We approach in this paper the structural evolution problems within components-based software architecture. We note that few ADLs integrate the evolution mechanism in their development process. To face this lack, we propose SAEV a model for software architecture evolution. SAEV considers each architectural element as first class element, which can be positioned on three abstraction levels: meta level, architectural level and application level. SAEV considers and specifies the evolution at these different levels in uniform way, this represents its first characteristic. The execution of the evolution operations is ensured through the selection and the triggering of the suitable evolution rules. These evolution rules described according to the ECA formalism represent the second characteristic of SAEV. These rules must also respect the invariants associated with the architectural elements in order to guarantee the consistency of the architecture. All evolution rules describing an architectural element evolution are gathered on its evolution strategy. By considering the various abstraction levels, SAEV is intended to be a generic, uniform, and an open model, allowing the description of the evolution at a high level of abstraction and therefore facilitating its reuse.

MOTS-CLES : architecture logicielle – modèle d'évolution – règles d'évolution
KEYWORDS: software architecture –evolution model – evolution rules

1. Introduction

Les architectures logicielles à base de composants prennent une place de plus en plus importante dans le domaine de l'ingénierie des systèmes. Elles permettent de décrire les systèmes comme un ensemble de composants en interaction. Les architectures logicielles ont pour motivation principale de réduire les coûts et les délais de développement des applications, de favoriser la réutilisation et l'évolution et de limiter la distance sémantique entre la conception et l'implémentation.

A l'inverse du monde industriel, qui propose des composants fortement liés à des serveurs, systèmes ou modèles propriétaires, l'approche académique s'intéresse à la formalisation de la notion d'architecture logicielle grâce à des langages de description d'architectures logicielles (dits ADL : Architecture Description Languages). Les ADLs offrent un niveau d'abstraction élevé pour la spécification et le développement des systèmes logiciels. Aujourd'hui, plusieurs ADLs sont définis pour aider au développement à base de composants, tels que C2[2], ACME[10], Rapide[13], UML2.0[18], UniCon[25], Metah[17], etc.

L'évolution dans les architectures logicielles est un mécanisme très important qui permet d'éviter que celles-ci ne restent figées et soient obsolètes par rapport aux besoins en perpétuels changements. En effet, cette évolution architecturale allonge la durée de vie des systèmes et ainsi garantit leur viabilité économique. Malgré les progrès constants sur les ADLs, l'évolution logicielle reste mal ou peu abordée par ces derniers. Pour ceux qui l'abordent, leurs propositions se limitent à des mécanismes souvent influencés par le langage de programmation support de l'implémentation tels que l'instanciation, le sous typage, la composition [17].

Nous proposons dans cet article un modèle d'évolution pour les architectures logicielles baptisé SAEV (Software Architecture EVolution model). Nous nous intéressons dans ce modèle à l'évolution de la structure d'une architecture logicielle. Nous nous focalisons dans un premier temps sur l'évolution statique de l'architecture avec une perspective d'utiliser le même modèle dans le cas d'évolution dynamique. SAEV propose un ensemble de concepts pour décrire et modéliser une évolution donnée ainsi que des concepts pour gérer l'exécution de cette évolution tout en sauvegardant la cohérence de l'architecture ayant évoluée.

Après cette introduction l'article sera structuré comme suit : la section 2 présente quelques définitions des concepts et approches d'évolution d'architectures logicielles existantes. La section 3 présente le modèle d'évolution proposé à travers son méta modèle, ses concepts et son mécanisme d'exécution. Cette section illustre aussi les principales étapes de la mise en œuvre du modèle SAEV. Nous terminons par une conclusion et des perspectives.

2. Architecture logicielle : concepts et approches d'évolution

Un modèle architectural d'un système fourni un modèle du système d'un niveau d'abstraction élevé en termes de composants qui réalisent les fonctionnalités et des connecteurs qui relient ces composants [14] et coordonnent leurs interactions afin de satisfaire des contraintes globales [3]. Ainsi, pour spécifier les modèles architecturaux,

plusieurs langages de description sont proposés, nous présentons dans ce qui suit leurs concepts les plus communément admis.

2.1. Concepts de base des modèles architecturaux

Les architectures étant au centre des préoccupations des phases de conception et de développement, des méthodes et notations formelles se devaient d'être proposées pour les spécifier de façon conceptuelle. Ainsi, nous pouvons définir un ADL comme un langage qui fournit des caractéristiques pour obtenir un modèle architectural d'un système logiciel. Plusieurs ADLs ont été alors proposés tel que : C2[2], ACME[10], Rapide[13], UML2.0[18], UniCon[26].

Les concepts de base d'un modèle architectural sur lesquels s'accordent l'ensemble des travaux portant sur les ADLs [1,18] sont : (1) les *composants*, (2)les *connecteurs*, et les (3) *configurations architecturales*.

Composant : un composant est une unité de calcul ou de stockage à laquelle est associée une unité d'implantation. A un composant est associé une interface qui décrit les points d'interaction entre le composant et son environnement extérieur. L'interface du composant indique les services (messages, opérations et variables) que le composant fournit, ainsi que les services requis à partir d'autres composants dans l'architecture.

Connecteur : un connecteur est un bloc de constructions architecturales utilisé pour modéliser les interactions entre les composants et les règles qui régissent ces interactions. Un connecteur peut représenter une interaction simple comme un appel d'une procédure, comme il peut représenter une interaction complexe comme une requête SQL entre une base de données et une application [11]. Les connecteurs ne sont pas toujours considérés comme des entités de première classe dans tous les ADLs, ils sont implicites dans certains ADLs.

Configuration : une configuration représente un graphe de composants et de connecteurs. Elle définit comment les composants sont reliés entre eux à l'aide de connecteurs. Cette notion est nécessaire pour déterminer si les composants sont bien reliés, leurs interfaces s'accordent, etc. Une configuration est décrite par une interface qui lui permet de communiquer avec d'autres configurations, ainsi qu'avec ses composants internes.

2.2. Niveaux d'abstraction d'une architecture logicielle

Tous les ADLs considèrent le composant comme une entité de première classe. Ils distinguent ainsi le composant-type de ses composants-instances. Ceci n'est pas le cas des autres concepts tels que les connecteurs, les configurations, etc. Pour notre part, nous considérons chaque concept admis par les ADLs comme élément architectural de première classe. Chaque élément architectural peut être positionné sur l'un des trois niveaux d'abstraction : le *niveau Méta*, le niveau *Architectural* et le *niveau Application* (figure 1). Nous jugeons nécessaire de réifier les éléments architecturaux afin de pouvoir les représenter et les manipuler et, a fortiori, les faire évoluer.

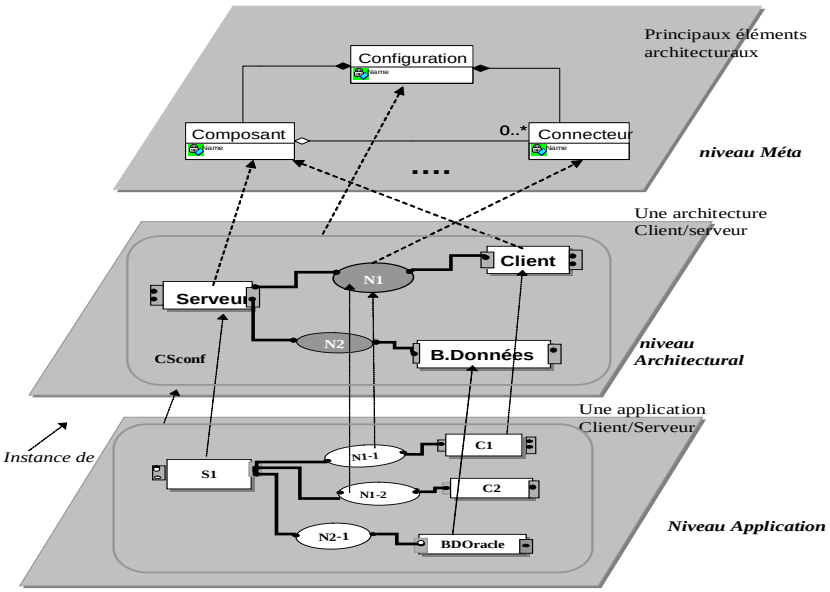


Figure 1: Les niveaux d'abstraction d'une architecture logicielle

Niveau Méta : c'est le niveau de définition de tous les éléments architecturaux qu'un ADL peut proposer pour la description d'une architecture logicielle. Nous considérons les plus communs : configuration, composant, connecteur ...

Niveau Architectural : ce niveau permet de décrire une architecture en utilisant une ou plusieurs instances des éléments architecturaux du niveau méta. La figure précédente présente une architecture Client/Serveur composée d'une *Configuration type* `CSCon` ; de trois *composants types :* `client`, `serveur` et `B.données` et de deux *connecteurs* type `N1`, `N2`.

Niveau Application : à ce niveau une ou plusieurs applications peuvent être construites conformément à leurs architectures décrites au niveau *architectural*. Chaque élément architectural du niveau application est une instance d'un élément architectural du niveau architectural. Par exemple à partir de l'architecture client/serveur précédente on peut construire l'application suivante composée de la configuration *`Cf`* : instance de `CSConf;` de deux composants *C1, C2 :* instances du composant `client ;` du composant `BDoracle :` instance du `composant B.données` ; `S1 :`instances du composant `serveur;` de deux connecteurs *`N1-1, N1-2 :`* instance du connecteur *`N1`*.

Nous avons présenté dans cette section les architectures logicielles au travers de leurs concepts de base et leurs différents niveaux d'abstraction. Nous avons mis l'accent sur les points que nous jugeons importants pour aborder la problématique de l'évolution dans les architectures logicielles.

2.3. Approches d'évolution dans les architectures logicielles

L'évolution dans les architectures logicielles est apparue comme une nouvelle préoccupation à laquelle la communauté des architectures logicielles doit répondre. Aborder l'évolution au niveau de l'architecture logicielle offre l'avantage de pouvoir travailler sur la structure globale du système en évitant ainsi d'intervenir directement sur le code source. Nous constatons que peu de propositions ont été faites par les ADLs pour l'évolution des architectures logicielles. Nous pouvons distinguer dans ces propositions deux catégories d'évolution : évolution statique et évolution dynamique. Nous apportons dans ce qui suit une brève présentation de ces deux catégories.

Evolution statique : elle est réalisée à la spécification ou à la compilation du système qu'elle décrit. La spécification de l'évolution statique est réalisée souvent grâce à des concepts et mécanismes supplémentaires intégrés dans l'ADL qui a servi à la description initiale de l'architecture. Ces concepts et mécanismes s'appliquent aux éléments architecturaux tels que les composants, les connecteurs et les configurations. Parmi ces mécanismes on peut citer : l'instanciation, l'héritage et le sous-typage, la composition [19, 20].

Evolution dynamique : signifie la possibilité d'introduire des modifications dans le système durant son exécution. Elle est la résultante de la prise en compte des changements structurels ou comportementaux, pouvant intervenir durant l'exécution d'une architecture et qui opèrent sans interruption du système. Ces changements dynamiques peuvent être prévus ou non au moment de la spécification de l'architecture, on parle alors d'évolution dynamique *planifiée* et d'évolution *non planifiée*. Dans le cas de l'évolution planifiée, tous les changements susceptibles de survenir à l'exécution du système doivent être connus au préalable, et doivent ainsi être spécifiés dans la description de l'architecture. Dans le cas de l'évolution non planifiée, les changements susceptibles de survenir ne sont pas connus à l'avance. Ils doivent opérer au moment de l'exécution du système : c'est l'évolution la plus complexe à gérer.

Nous avons réalisé une analyse des différents mécanismes proposés par les ADLs dans le cas de chaque catégorie d'évolution [25]. Nous constatons que dans le cas de l'évolution statique et de l'évolution dynamique planifiée, la spécification de l'évolution est réalisée par l'ADL qui a servi à la spécification de l'architecture. Ainsi, la gestion de l'évolution est imbriquée dans la spécification de l'architecture, donc difficile à distinguer et surtout à réutiliser. Nous pouvons constater aussi que certains ADLs mettent l'accent sur l'évolution des composants et non sur les autres éléments architecturaux, ce qui limite l'évolution d'une architecture. En ce qui concerne l'évolution non planifiée, elle reste l'évolution la plus difficile à mettre en œuvre vue la difficulté de gérer les impacts automatiquement et sans incohérences, le nombre de propositions à notre connaissance est très restreint.

Afin d'aider à faire face à la problématique de l'évolution des architectures logicielles à base de composants, nous proposons un modèle générique d'évolution baptisé SAEV : Software Architecture EVolution Model.

3. SAEV : un modèle d'évolution des architectures logicielles

SAEV propose une solution à la problématique de l'évolution structurelle dans les architectures logicielles et plus précisément à l'évolution structurelle de ces dernières. SAEV doit permettre de décrire et de modéliser l'évolution d'une architecture logicielle, ainsi que de gérer l'exécution de cette évolution, indépendamment de son langage de description. Pour cela il vise à permettre :

- l'abstraction de l'évolution, en la distinguant explicitement du comportement propre à tout élément de l'architecture. Les avantages sont :
  - d'offrir des mécanismes pour décrire et gérer cette évolution indépendamment du comportement des éléments architecturaux et de leurs langages de description ;
  - de permettre une spécification de l'évolution générique et réutilisable
- l'évolution de tous les éléments architecturaux (composant, connecteur, configuration…) à n'importe quel niveau d'abstraction.
- la définition des opérations d'évolution, voire les mêmes opérations d'évolution pour faire évoluer tous les éléments architecturaux. SAEV est ainsi générique et uniforme.
- la gestion des impacts d'une évolution d'un élément architectural. Pour cela, SAEV doit définir des règles d'évolution et gérer leur exécution.
- l'évolution aussi bien statique (lors des spécifications de l'architecture) et à terme l'évolution dynamique (lors de l'exécution de l'application).

3.1. Description de SAEV

Pour répondre aux objectifs fixés précédemment, nous considérons que l'évolution d'une architecture logicielle se reflète par les différents changements dans sa structure et/ou dans celle de ses éléments constitutifs. Un changement dans une architecture est toujours engendré par une ou plusieurs opérations appliquées à cette architecture ou à l'un de ses éléments. Parmi ces opérations possibles, nous citerons à titre d'exemple la suppression, la substitution et la modification d'un élément. Ainsi, non seulement l'opération invoquée selon le contexte doit être identifiée, mais ses éventuels impacts également doivent l'être pour éviter d'introduire des incohérences dans l'architecture ayant évoluée. Ainsi, le modèle d'évolution proposé doit répondre au minimum aux questions suivantes :

- quels sont les éléments d'une architecture amenés à évoluer ?
- quelles sont toutes les opérations agissant sur ces éléments ?
- quels sont les impacts d'une opération sur un élément et comment les gérer ?

En se basant sur ces préoccupations et sur les objectifs fixés précédemment, SAEV offre des concepts pour décrire et gérer l'évolution de l'architecture ainsi qu'un mécanisme décrivant le processus opératoire à suivre pour exécuter une évolution donnée. Ces concepts sont décris par le méta modèle (selon le langage UML[6]) suivant:

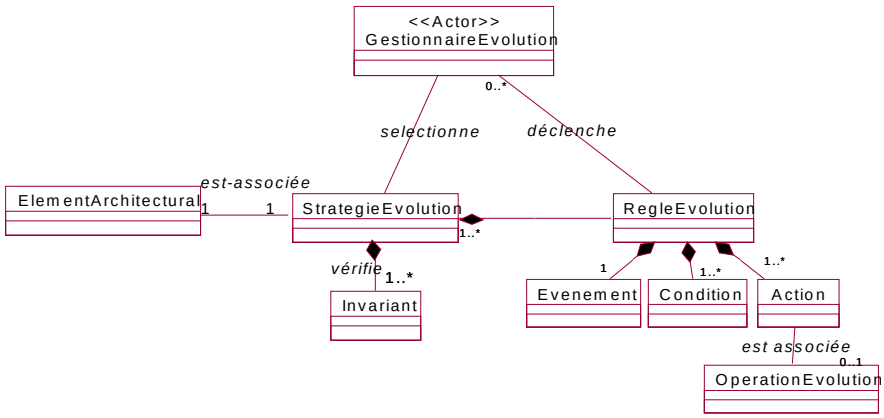


Figure 2: Méta modèle de SAEV

Le concept d'*élément architectural* représente tout élément de première classe de l'architecture qui peut être amené à évoluer. Pour décrire l'évolution d'une architecture, une *stratégie d'évolution* est associée à chaque élément architectural. Une stratégie regroupe l'ensemble de *règles d'évolution* nécessaires pour mener les *opérations d'évolution* sur l'élément architectural auquel elle est appliquée. Ainsi chaque règle d'évolution doit respecter les invariants définis sur l'élément architectural.

3.1.1. Concepts de SAEV

a. Elément architectural : il représente tout élément de première classe dans une architecture logicielle. A chaque niveau d'abstraction un *élément architectural* peut être aussi bien une configuration, un composant, un connecteur, une interface ou tout autre concept supporté par son ADL. Par exemple au niveau méta l'*élément architectural* peut représenter le concept « Composant », au niveau architectural il peut représenter le « Composant Client », et au niveau application il peut représenter le « Composant Client X ». Nous avons introduit ce concept dans SAEV pour modéliser et réifier l'élément de l'architecture logicielle à faire évoluer.

b. Invariant : un invariant représente une contrainte sur un élément architectural ou sur l'ensemble de l'architecture. Un invariant doit être respecté tout au long de son cycle de vie, et cela quelle que soit l'opération d'évolution appliquée à cet élément. Tout changement dans l'architecture logicielle doit maintenir la vérification de chaque invariant pour garantir la cohérence globale de l'architecture. Il est également à noter que les invariants peuvent être définis à tout niveau d'abstraction. Les invariants sont définis dans un niveau, et ils doivent être respectés lors de l'évolution du niveau inférieur. Nous illustrons avec les invariants suivants :

- au niveau méta on trouvera des invariants relatifs à la définition des concepts (selon l'ADL considéré). Nous pouvons citer à titre d'exemple les invariants suivants relatifs à la configuration: une configuration doit être composée au moins d'un composant ; un connecteur dans une configuration doit être relié au minimum à deux composants ;

- au niveau architectural, on peut définir des invariants relatifs à un type d'architecture donnée. Par exemple si on considère l'architecture client serveur (figure 1) on peut définir les invariants suivant : un connecteur entre un composant *client* et un composant *serveur* doit être de type *RPC* ; le nombre de client relié à un serveur ne doit pas dépasser 30.

c. Opération d'évolution : une opération d'évolution représente toute opération que l'on peut exécuter sur un élément architectural pour le faire évoluer ou faire évoluer l'architecture qui le contient. Les opérations minimales que nous avons recensées et que l'on peut exécuter sur un élément architectural, à n'importe quel niveau d'abstraction sont : l'ajout, la suppression, la modification et la substitution. Nous donnons quelques exemples d'application de ces opérations sur les éléments architecturaux du *niveau méta.* Opération d'évolution de la *configuration* sont :

- Ajout/ suppression/substitution d'un composant.
- Ajout / suppression/ substitution d'un connecteur.
- Modification d'un nom de composant ou de connecteur

d. Règles d'évolution : la *règle d'évolution* est le concept le plus important de SAEV. Elle décrit l'exécution d'une opération d'évolution sur un élément architectural. La règle d'évolution doit préciser son événement déclencheur, les conditions nécessaires à satisfaire ainsi que l'exécution de sa partie active qui représente les changements à faire sur l'élément architectural concerné ainsi que la propagation d'impacts. Cette dernière se fait au regard des invariants définis et selon les stratégies et donc des règles d'évolution existantes. Nous avons choisi de modéliser ces règles d'évolution par le formalisme ECA (Evénement/ Condition/ Action), que nous jugeons adapté pour montrer le déclenchement et l'exécution automatique de ces règles d'évolution. Ce formalisme servira aussi pour exprimer et assurer la maintenance de la cohérence de l'architecture à faire évoluer, notamment par la propagation d'impacts. Ainsi, chaque règle d'évolution est composée :

- d'un *événement* : qui représente le message d'invocation reçu de l'environnement (le concepteur ou par une autre règle suite à son exécution), vers l'élément à faire évoluer.
- d'une ou plusieurs *conditions* : qui doivent être satisfaites pour exécuter la partie action d'une règle d'évolution. Ces conditions doivent respecter les invariants de l'élément sur lequel l'opération est invoquée.
- d'une ou plusieurs *actions* à exécuter, une action peut être :
  - un événement, dans ce cas il sera redirigé vers une autre règle ;
  - une opération à exécuter sur un élément architectural. Nous la notons : *nom-element.Execute.nom-operation(paramétres)*;

A chaque niveau d'abstraction nous définissons un ensemble de règles d'évolution, qui seront mémorisées dans une base de règles. SAEV offre aussi la possibilité au concepteur de définir ses propres règles d'évolution. Nous donnons dans la suite un exemple d'une règle d'évolution définie au *niveau méta*, cette règle décrit un cas de suppression d'un composant d'une configuration. Cet exemple est volontairement simpliste. L'objectif est d'illustrer la description d'une règle :

R1 : suppression d'un composant C de sa configuration Cf **Evènement :** supprimer-composant(**C**: composant, **Cf**: Configuration); **Condition: C** ∈ comp(**Cf**), interf-requi (**C**) reliée à Interf-requi(**Cf**), ∃ **NC**⊂ connect(**Cf**), et ∀ **N** ∈ **NC** **N** est relié à **C** et **N** non partagé **Action:** pour tout **N**∈ **NC** Supprimer-connecteur (**Cf**,**N**) Pour b ∈ bindings(**Cf**,**C**) supprimer-binding(**Cf**,**C**,**b**) Pour I ∈ interface-comp(**C**) supprimer-interf-comp(**Cf**,**C**,**I**) Cf.execute-supprimer-composant(**C**).

- *comp(**Cf**)* : l'ensemble des composants de la configuration **Cf** ;
- *connect(**Cf**)* : l'ensemble des connecteurs de la configuration **Cf** ;
- *binding(**Cf,C**)* : l'ensemble des liaisons entre l'interface configuration **Cf** et l'interface du composant **C**

La règle *R1* décrit la suppression du composant **C** qui se trouve à l'extrémité de la configuration **Cf.** Elle sera déclenchée par l'arrivée de l'événement : *supprimer-composant(**Cf**: Configuration, **C**: composant),* le nom du composant à supprimer, ainsi que la configuration a laquelle il appartient sont passés en paramètre. Les trois premières actions de *R1* correspondent à des événements qui déclenchent d'autres règles. Ces règles seront exécutées d'une façon séquentielle. La dernière action de *R1* déclenche la suppression proprement dite du composant *C*.

e. Stratégie d'évolution

Une stratégie d'évolution est associée à un élément architectural. Une stratégie regroupe l'ensemble des règles d'évolution d'un élément architectural. Ces règles peuvent être des règles déjà définies dans la base de règles comme elles peuvent être des règles nouvellement définies par le concepteur. A la réception d'un événement d'évolution par un élément architecturale, le gestionnaire d'évolution retrouve la règle d'évolution à déclencher en intérégeant la stratégie d'évolution associée à cet élémentarchitectural.

f. Gestionnaire d'évolution: tous les concepts précédemment définis ont un rôle descriptif. Le gestionnaire d'évolution quant à lui est chargé de gérer l'évolution d'une architecture logicielle donnée au vu de ces concepts. Son rôle est d'intercepter les événements émanant du concepteur ou des règles d'évolution vers l'architecture ou l'un de ses éléments, puis d'identifier la stratégie d'évolution correspondante. Suivant cette stratégie, il recherche la ou les règles d'évolution éligibles (selon l'événement déclencheur et la satisfaction de la partie condition) qu'il déclenche, tout en vérifiant la validité des invariants à la fin de l'évolution.

3.1.2. Mécanisme d'évolution

Le mécanisme d'évolution décrit le processus opératoire à suivre pour faire évoluer une architecture logicielle. Ce processus est décrit par le diagramme de séquences d'UML suivant :

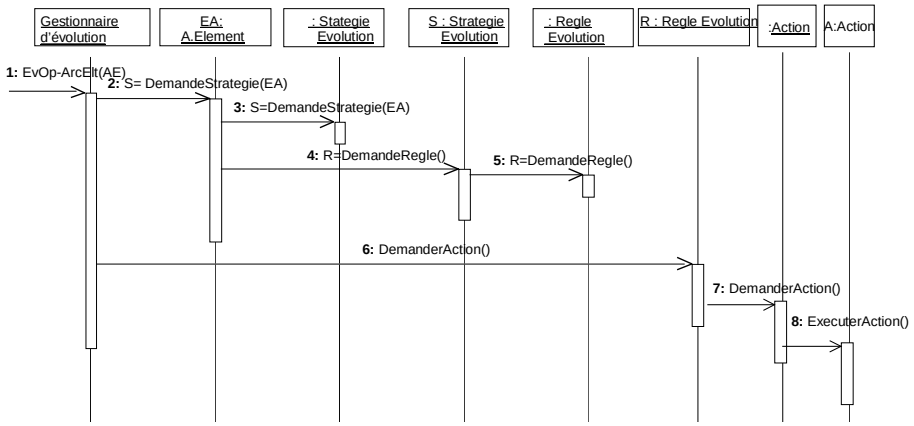


Figure 3: Processus opératoire de SAEV

L'évolution est déclenchée automatiquement à la réception d'un événement, le gestionnaire d'évolution exécute séquentiellement les étapes suivantes:

- **1 :** intercepte chaque événement émit vers l'élément architectural, soit par le concepteur soit par une règle d'évolution ;
- **2 :** et **3 :** sélectionne la stratégie d'évolution associée à l'élément architectural sur lequel l'événement est invoqué ;
- **4 :** et **5 :** sélectionne ensuite dans cette stratégie la ou les règles d'évolution correspondantes à l'événement et dont leurs parties condition sont satisfaites (par raison d'espace ,nous n'avons pas illustré dans le figure l'évaluation de l'événement et des conditions).
- **6,7, 8 :** déclenche la partie action de la règle sélectionnée. Deux cas peuvent se présenter :
    - si l'action correspond à un événement, le gestionnaire l'intercepte aussi et repart alors de l'étape 1.
    - s'il s'agit d'une autre action (opération élémentaire) le gestionnaire déclenche alors son exécution sur l'élément architectural sur lequel l'opération est invoquée.

Nous avons illustré le processus d'exécution sur un cas d'évolution sans conflit (le gestionnaire d'évolution a identifié une seule règle puis il a déclenché son exécution). A la fin de chaque évolution le gestionnaire d'évolution lance la vérification des invariants. Si des incohérences sont détectées, celles ci sont renvoyées au concepteur qui peut les rectifier en lançant d'autres règles d'évolution. Dans le cas contraire le gestionnaire d'évolution annule toutes les évolutions qui ont conduit aux incohérences.

3.2. SAEV et les niveaux d'abstraction

SAEV définit un ensemble de concepts pour modéliser d'une façon explicite l'évolution des architectures logicielles au niveau architectural et au niveau application. En effet, il peut être positionné au niveau méta (les règles d'évolution, stratégies d'évolution et invariants seront définit sur les éléments architecturaux du niveau méta) pour gérer l'évolution du *niveau architectural*. Il peut être positionné au *niveau*

architectural (tous ses concepts seront définis sur les éléments architecturaux du niveau architectural) pour gérer l'évolution du *niveau application*. Ceci est illustré dans la figure suivante (5.a) :

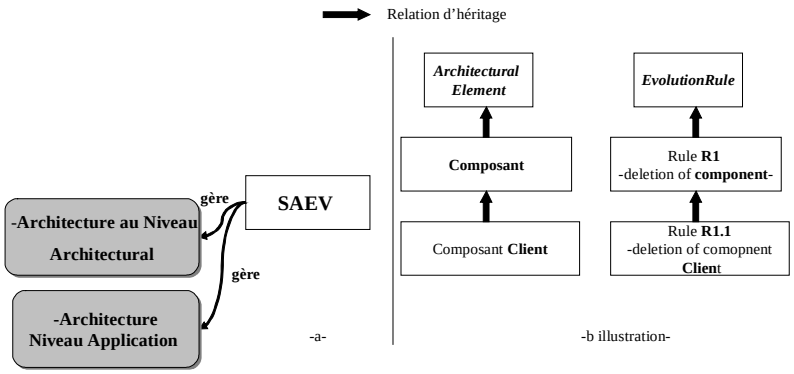


Figure 4: SAEV et les niveaux d'abstraction

A titre d'exemple (figure 5.b), si on considère une architecture *client*/*serveur* on peut définir une règle R1 au niveau méta (règle de suppression d'un composant), cette règle sera invoquée lors de la suppression de n'importe quel composant au niveau architectural. Cette règle R1 peut être ensuite spécialisée par une autre règle d'évolution R1.1 (suppression d'un composant client) au niveau architectural, qui sera invoquée lors d'une suppression de n'importe quel composant de type client au niveau application. Cette règle R1.1, en plus de ce qui est spécifié par R1, précise ce qui est spécifique à la suppression d'une instance d'un composant Client.

3.3. Mise en œuvre de SAEV

La mise en œuvre de SAEV est en phase de prototypage. Cette première mise en œuvre a suivi deux étapes : la première étape était consacrée à la spécification architecturale de SAEV en UML2.0 et la deuxième étape à l'implémentation de cette spécification en Java.

3.3.1. Spécification architecturale de SAEV en UML2.0

SAEV est également un modèle réflexif. Il est en mesure de gérer sa propre évolution. Pour cela, un premier travail a été de spécifier SAEV comme une architecture à base de composants.

Pour réaliser la spécification architecturale de SAEV, nous avons opté pour UML2.0 [24]. UML étant à la base un langage de modélisation objet, les adaptations de sa dernière version 2.0 permettent la représentation d'architectures logicielles. Nous avons fait un travail préliminaire de choix de traduction de chaque concept de SAEV en un concept architectural UML2.0 adéquat. Le résultat de cette traduction est illustré dans la spécification suivante de SAEV :

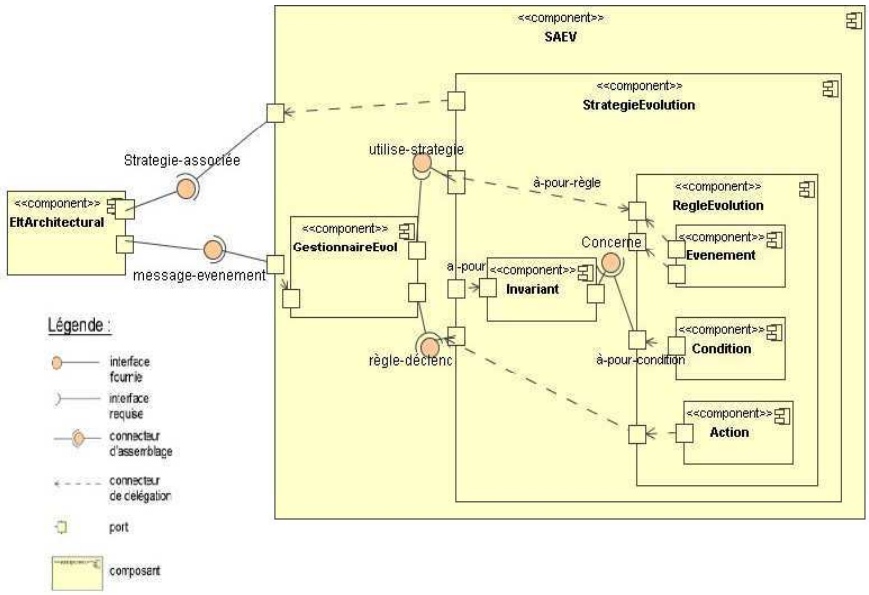


Figure 5: Description de SAEV en UML2.0

Nous avons voulu uniformiser la spécification architecturale de SAEV pour deux raisons : SAEV s'applique sur des architectures logicielles à base de composants ; il est représenté en tant que telle ; SAEV se décrit comme une architecture logicielle et qui peut être appelé à évoluer, notamment pour prendre en compte ses futures extensions et adaptations. Pour cela chaque concept du méta modèle SAEV (section 3.1) est modélisé par un composant fournissant et requérant des services. La communication entre les composants se fait par le biais de connecteurs d'assemblage. Les relations de composition du méta modèle de SAEV sont représentées dans la figure par une imbrication de composants. Les liens entre les composites et leurs composants sont décris par des connecteurs de délégation (bindings).

Par soucis de clarté nous n'avons pas mis tous les détails de la description architecturale dans la figure. Nous détaillons juste à titre d'exemple un seul composant : le **gestionnaire d'évolution** (figure 6).

Le gestionnaire d'évolution joue un rôle important dans le mécanisme d'évolution proposé par SAEV. Il prend les décisions, décide d'entamer une évolution, de la propager mais aussi de l'arrêter en cas de conflit. C'est pour cette raison que beaucoup de connexions sont établies avec les autres composants. Le composant *gestionnaire d'évolution* dans la figure 5 est relié à la configuration *SAEV* par le biais du connecteur de délégation e*vent-evol*. Il est relié au composant stratégie par les connecteurs d'assemblage *utilise-stratégie* et *precedence-2*. Les services qui transitent via ces connecteurs sont décris par le tableau suivant :

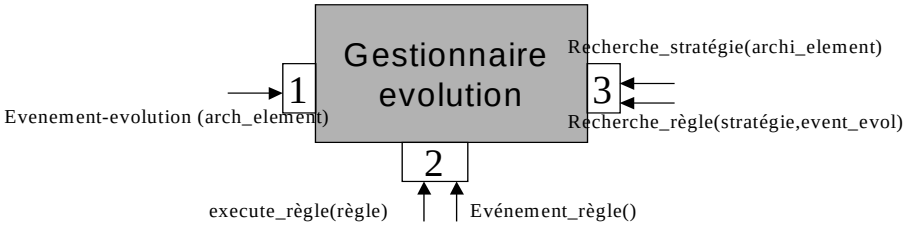


Figure 6: Composant gestionnaire d'évolution

Tableau 1: Description du composant gestionnaire d'évolution

port	Service	Description
Event-evol	Evenement-evolution(archi-element)	Requis :réception de l'événement d'évolution de l'élément architectural à modifier
Utilise-strategie	Recherche-stratégie(archi-element)	Requis : recherche d'une stratégie pour un élément considéré.
	Recherche-règle(stratégie, event-evol)	Requis : recherche de règles pour une stratégie et un événement
Precedence-2	Evenement-regle()	Requis : réception d'un événement de propagation provenant du composant **Action**
	Execute-regle(regle)	Requis : lance l'exécution d'une règle d'évolution donnée

3.3.2. Implémentation de SAV : de UM2.0 vers Java

Après la description conceptuelle de SAEV, le but maintenant est d'obtenir un prototype exécutable de SAEV dans le langage orienté objet Java. Nous disposons déjà de la description de SAEV en UML2.0, mais comme ce dernier ne permet pas une génération automatique du code, nous avons définis des règles de passage de UML2.0 vers un langage Java que nous résumons comme suit :

Chaque composant et chaque connecteur sont représentés par une classe Java. A chaque classe est associée ensuite une interface Java à implémenter pour fournir ou requérir des méthodes. Le choix a été fait de modéliser le lien entre composant et connecteur grâce aux références Java. Toutefois, la particularité ici réside dans le fait que le sens de direction du connecteur ou de la restitution des services communiqués, conditionnent l'ajout de références dans les classes Java. Ainsi, un **Composant** contient une référence du connecteur lui communiquant un service requis, et inversement le **Connecteur** contient la référence du **Composant** fournissant le service en question. De la même manière, un **Composant** comporte les références de ceux qui lui fournissent des services, sans passer par des connecteurs (*binding*).

Nous présentons juste à titre d'exemple le composant ***RègleEvolution*** de la figure 7 (le composant qui représente le noyau de SAEV)**.** Un choix d'implémentation a été fait pour les parties Action et Condition des règles ECA. En effet, nous modélisons ces deux parties comme illustrée sur la figure suivante :

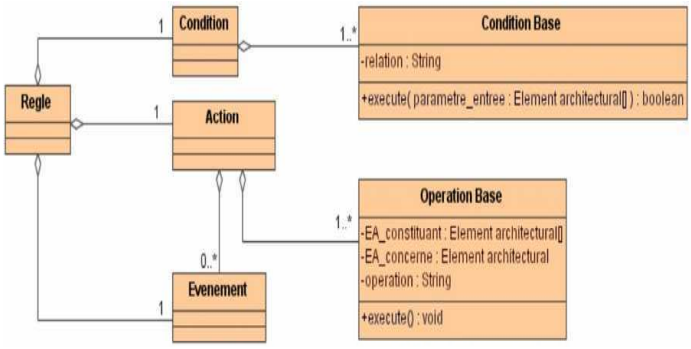


Figure 7: Modélisation d'une règle d'évolution

La partie *Condition* d'une règle comporte des *conditions de base*. Celles-ci ont un attribut *relation*, renseignant sur le type de la condition, exemple : l'appartenance à une configuration ou encore l'existence d'interface pour un composant. La méthode *execute()* est invoquée pour vérifier la condition sur laquelle elle est appelée. Son paramètre d'entrée constitue l'ensemble des éléments architecturaux nécessaires pour effectuer la vérification de la condition. Alors, leur ordre repose sur le niveau hiérarchique : la configuration en premier, le composant, le connecteur et enfin l'interface architecturale. Fondée sur le même principe, la partie Action d'une règle ECA contient des opérations de base, ayant chacune les attributs suivants :

. **EA_concerne** qui est l'élément architectural concerné par l'opération, exemple : la *configuration* pour la *suppression d'un composant.*

. **EA_ constituant** qui regroupe l'ensemble des éléments architecturaux sur lesquels sont effectués les opérations d'évolution, exemple : la liste des connecteurs pour la suppression de ceux connectés à un composant en cours de suppression.

. **Opération** est l'attribut informant sur le type de l'opération. La réalisation d'une opération de base se fait grâce à l'exécution de la méthode *execute()*. Celle-ci effectue un travail en fonction de l'attribut *operation* de l'objet sur lequel est appelée la méthode.

3.4. Bilan

Le développement du prototype de SAEV en Java, à nécessiter beaucoup d'efforts de décisions concernant la modélisation des éléments architecturaux. La traduction ne semble pas toujours évidente, néanmoins ce prototype nous a permet de simuler l'exécution des règles d'évolution, et le déclenchement automatique des autres règles d'évolution permettant la propagation de l'impact d'une évolution donnée.

4. Conclusion et perspectives

Nous avons proposé dans cet article un modèle d'évolution pour les architectures logicielles baptisé SAEV. Le modèle SAEV offre des concepts pour décrire l'évolution d'une architecture au travers celle de ses éléments, et des concepts permettant de gérer l'exécution de cette évolution. Nous avons considéré chaque concept admis par les ADLs comme un élément architectural de première classe, qui peut être positionné sur

un niveau d'abstraction. A chaque élément sont également associées des opérations d'évolution, chacune étant décrite au travers d'une ou plusieurs règles d'évolution. Ces règles doivent respecter les invariants associés aux éléments architecturaux pour éviter d'introduire des incohérences lors de l'évolution. L'ensemble des règles décrivant toutes les évolutions d'un élément architectural sont regroupées dans une stratégie d'évolution qui lui est associée.

SAEV répond à la majorité des objectifs décrit section 3. L'évolution est décrite explicitement et indépendamment de tout ADL ou langage de programmation. SAEV et générique et uniforme, il offre les concepts et mécanismes pour l'évolution de chaque élément architectural du niveau architectural et du niveau application. La spécification d'une évolution peut être réutilisée, notamment les règles et stratégies qui la décrivent. Nous n'avons pas abordé dans cet article l'application de SAEV pour l'évolution dynamique d'une architecture logicielle.

Nous avons développé un prototype de SAEV en langage Java, en passant par une spécification architecturale en UML2.0 la traduction des concepts architecturaux en Java n'était évidente mais nous avons défini certaines conventions, qui ont permis ce passage.

L'une des perspectives de ce travail est d'étudier l'apport de SAEV dans le cas de l'évolution dynamique. Cela s'appuiera sur la définition d'autres invariants et d'autres règles d'évolution spécifiques à l'aspect dynamique d'une architecture logicielle. Nous envisageons d'introduire ensuite d'autres opérations avancées telles que la fusion, la décomposition, et surtout le versionnement qui est une opération importante pour les systèmes souhaitant garder la trace de leurs évolutions. Nous traiterons ces opérations avec le même principe que les autres opérations de base déjà définies.

5. Bibliographie

1. Project ACCORD: Etat de l'art sur les Langages de Description d'Architecture (ADLs) , Rapport technique, INRIA, France, Juin 2002.
2. R. Allen, R. Douence, D.Garlan: Specifying and Analyzing Dynamic Software Architectures, In proceeding of the Conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal Mars 1998.
3. R. Allen and D. Garlan, "Formal Connectors", CMU-CS-94-115, Carnegie Mellon University, March 1994.
4. L F. Andrade, J.L. Fiadeiro : Architecture based evolution of software systems, LNCS 2804 : 148-181, 2003.
5. O. Barais, L. Duchien, AF. Le Meur: A Framework to Specify Incremental Software Architecture Transformations. EUROMICRO-SEAA 2005: 62-69
6. G. Booch, J. Rumbaugh, and I. Jacobson: The Unified Modeling Language User Guide, Addison-Wesley Professional, Reading, Massachusetts, 1998.
7. F. Duclos, J.Estublier and R. Sanlaville : Architectures Ouvertes pour l'Adaptation des Logiciels. Revue Génie Logiciel, N°58, pp:19-25, Septembre 2001
8. W.B. Frakes, A case study of a reusable component collection in the information retrieval domain, The Journal of Systems and Software, pp 265-270, 2004.
9. D. Garlan: Software Architecture. In J. Marcinik, editor, Encyclopedia of Software Engineering. John Wiley & Sons, Inc., 2001.

10. D. Garlan, R. Monroe, D. Wile: ACME: Architectural Description Of Components-based systems, Leavens Gary and Sitaraman Murali, Foundations of component-Based Systems, Cambridge University Press, 2002,pp. 47-68.
11. D. Garlan, D. Perry: introduction to the special issue on software architecture, IEEE Transactions on Software Engineering, 21(4), April 1995.
12. R. Land: An architectural approach to software evolution and integration, Licentiate thesis, ISBN 91-88834-09-3, Department of Computer Engineering, Mälardalen University, September 19th 2003.
13. D. Luckham, M. Augustin, J. Kenny, J. Vera, D. Bryan, W. Mann: Specification and analysis of System Architectures using Rapide, IEEE Transaction on Software Engineering, vol.21, N° 4, April 1995, pp.336-355.
14. J. Magee, N Dulay, S. Eisenbach, J. Kramer: Specifying Distributed Software Architectures, In Proceedings of the fifth European Software Engineering conference, Barcelona, Spain, September 1995.
15. N. Medvidovic: ADLs and Dynamic Architecture Changes. In second International Software Architecture Workshop (ISAW-2), San Francisco, 1996.
16. N. Medvidovic, D.S.Rosenblum, and R.N. Taylor: A Language and Environment for Architecture-based Software Development and evolution. In proceeding of the 21st international conference en Software engineering , pp.44-53, may 1999.
17. N. Medvidovic, R. N. Taylor: A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering, Vol. 26, 2000.
18. OMG, UML 2.0 infrastructure specification, Technical Report ptc/03-09-15,Object Management Group (2003).
19. M. Oussalah, Changes and Versioning in complex objects, International Workshop on Principles of Software Evolution, IWPSE 2001, Sep. 10 & 11, Vienna University of Technology, Austria.
20. M. Oussalah: Ingénierie des composants : concepts et techniques, collectif sous la direction de M. Oussalah, Vuilbert informatique, 2005.
21. P. Oreizy : Decentralized Software Evolution; International Work on the principles of Software Evolution (IWPSE); Kyoto, Japan, 20-21 April 1998.
22. D. Pemberton, RComponents – Reflective & Adaptable Components, REFLEXProject, 2002.
23. D.E. Perry, A.L. Wolf : Fondations for study of software Architecture, In ACM/SIGSOFT Software Engineering Notes, volume 17, pages 40-52, october 1992
24. R. Roshandel, A.V.D. Hoek, M. Miki-Rakic, N. Medvidovic : Mae-A System Model and Environment for Managing Architectural Evolution : ACM Transactions on Software Engineering and Methodology, April 2004.
25. N.Sadou, D.Tamzalit, M. Oussalah, « How to manage uniformly Software Architecture at different abstraction levels », 24th ACM International conference on Conceptual Modeling (ER2005), pp. 16-30, 24-28 oct 2005., Klagenfurt, Autriche
26. M. Shaw, R. Deline, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnik: Abstractions for Software Architecture and tools to support them, IEEE Transactions on Software Engineering, pages 314-335, April 1995.
27. M.Shaw, D. Garlan: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.
28. Smeda, M. Oussalah, T. Khamaci : A Multi-Paradigm Approach to Describe Complex Software System", WSEAS Transactions on Computers, Issue 4, Volume, 3, October 2003, pp. 936-941.

Model driven engineering method for SAIA architecture design

Julien DeAntoni — Jean-Philippe Babau

CITI laboratory – INSA-Lyon
21, avenue Jean Capelle, 69621 Villeurbanne cedex
julien.deantoni ; jean-philippe.babau@insa-lyon.fr

ABSTRACT. *SAIA is an architectural style for the development of systems dedicated to process control. Designing an architecture that conforms to a style implies the manipulation of a lot of entities and the respect of numerous constraints. The approaches based on models and models transformations are well adapted to manage the complexity and to enforce the separation of concerns. This paper presents a model driven engineering method for the development and the validation of systems that conform to SAIA. Moreover, a tool supporting the method allows a systematic use of models and transformations.*

RÉSUMÉ. *SAIA est un style architectural destiné au développement de systèmes dédiés au contrôle de procédés. Construire une architecture conforme à un style architectural donné nécessite la manipulation de nombreuses entités et le respect de nombreuses contraintes. Les approches basées sur les modèles et les transformations de modèles permettent de gérer cette complexité et d'imposer une séparation des préoccupations. Notre objectif est alors de fournir une méthode basée sur les concepts définis par l'ingénierie dirigée par les modèles pour le développement et la validation de systèmes conformes à SAIA. Enfin, un outil implémente la méthode proposée afin d'aider le concepteur à respecter les modèles et leurs transformations.*

KEYWORDS: *MDE, real time, development method, architectural style, model validation, modeling tool.*

MOTS-CLÉS : *IDM, temps réel, méthode de conception, style architectural, validation de modèles, outil de modélisation.*

1. Introduction

Over the last few years, engineers have been faced with the problem of developing more and more complex real time systems in a world where time-to-market constraints are constantly increasing. Among real time systems, Systems Dedicated to Process Control (SDPC) are systems closely linked to the physical environment.

A SDPC builds, in real time, a view of the controlled process by using sensors (Cf. figure 1). Then it computes the necessary commands in order to control the process. Finally, commands are performed using actuators. Since the external environment has its own behaviors, all operations must be performed respecting real time constraints such as a minimum frequency or a maximum delay.

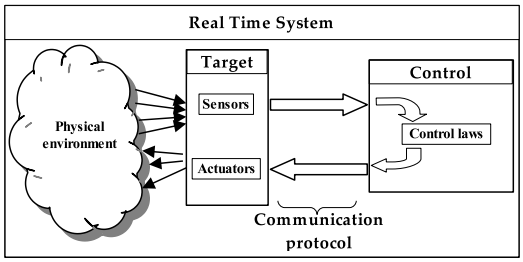


Figure 1. *Representation of a SDPC*

For these reasons, the SDPC communication part is developed in an ad-hoc and hand-made way. Of course, this type of development leads to concern merged code and specifications. In consequence, a software engineering approach has to be used to precisely specify the communication protocol; in a functional way as well as in an extra-functional way (Jumel, 2003). Among the various software engineering sub-domain, the software architecture gives propositions about the development and the early validation of software (Shaw *et al.*, 1996, Ghezzi *et al.*, 2000, Bass *et al.*, 1998).

The study of software architecture has highlighted that redundant configuration are used to solve classes of problems. These specific structures are identified as architectural style. SAIA[1] (DeAntoni *et al.*, 2005b) is an architectural style that has been defined for the development of SDPC. Its goal is to facilitate the development, the evolution and the timing analysis of the communication protocol part.

Building an architecture that conforms to a style implies the manipulation of a lot of entities and constraints. It severely increases the engineers' works and its complexity. So, to efficiently build and validate an architecture, methods and tools are necessary. As highlight by MDA (Model Driven Architecture : (OMG-MDA, 2003)), dealing with models and models transformations is a way to properly apply separation

1. SAIA : Sensor/Actuator Independent Architecture

of concerns, and so, to reduce complexity. The aim of this paper is to propose a model driven engineering method to support the conception of system with SAIA.

First, the section 2 presents an overview of SAIA. The next section describes the different models and models transformations implied during the conception of the system. Then, the section 4 presents a tool that implements the method. After, we present the related work and we give some conclusions and perspectives.

2. SAIA architectural style

2.1. *SAIA concepts*

Sensors Actuators Independent Architecture is an architectural style which defines entities and entity-associations for the modeling of SDPC. From a software engineering point of view, its goal is to separate the concerns between target, communication protocol and control (see figure 1). From a practical point of view, it allows a development of SDPC independently of specific sensors actuators; a realistic simulator-based simulation (DeAntoni *et al.*, 2005b) and an easy timing analysis (DeAntoni *et al.*, 2005a).

SAIA is a layered architecture. It distinguishes three layers through three models: one for the target, one for the communication protocol and one for the control. It allows the evolution of the control or the target without impacting each other. Each layer is specified thanks to components whose behaviors are accessible through interfaces.

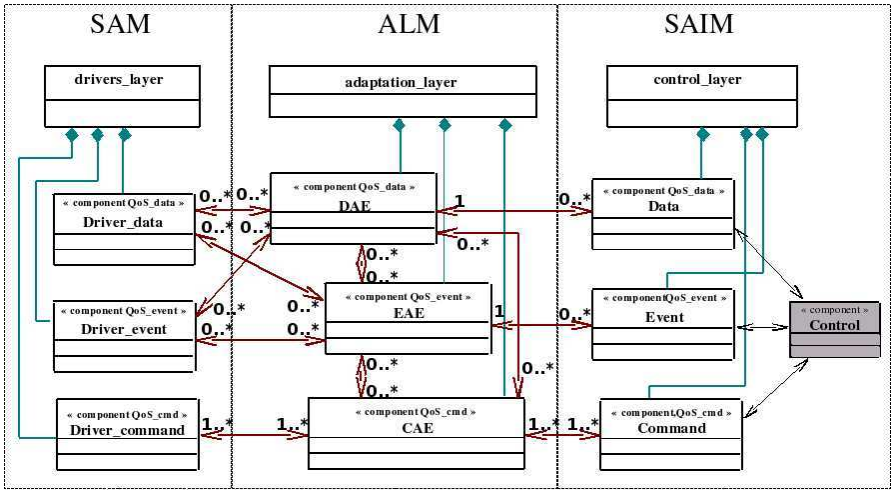


Figure 2. *SAIA layered organization*

2.2. *Control layer*

In the control layer, the control activity is based on a description of an ideal environment view called Input/Output (I/O). It models the state of the environment through data, events and the requirements on environment through commands. Each I/O describes the required QoS needed for the correctness of the control. There are three QoS characteristics : the arrival law, the delay and the precision (precision is not specified for an event). SAIA focuses on the interaction of a system with the external environment. Consequently, the control laws are seen in SAIA like black boxes. All these pieces of information are described in the SAIM (Sensors/Actuators Independent Model).

2.3. *Driver layer*

The driver layer is described by a model called SAM (Sensors/Actuators Model). It specifies the sensors and actuators services available on a specific platform. In the same way than the SAIM, there is a distinction between three entity types: driver_data, driver_event and driver_command. The two first entity types encapsulate a sensor driver whereas driver_command encapsulates an actuator driver. It provides a platform-specific view of the environment. Sensors and actuators drivers describe the QoS provided by the underlying hardware with the same QoS characteristics than the SAIM. To formally evaluate the provided QoS by sensors and actuators drivers is not a trivial activit but there are works on (Robles *et al.,* 1999, Ben-Hedia *et al.,* 2005). SAIA considers that the QoS provided is a priori known.

2.4. *Adaptation layer*

The adaptation layer, described by the ALM (Adaptation Layer Model) realizes the "smart" link (functional and extra-functional) between the ideal environment view of the SAIM and the platform specific environment view of the SAM. To do that, the adaptation layer is composed of three elements (Data Adaptation Element, Event Adaptation Element, Command Adaptation Element), each dedicated to a specific I/O type. Three[2] internal activities has been identified to specify an ALM element:

– `format`: it is used to cast driver_data or command value in order to manipulate them consistently. It consists in unity change and/or reference mark change.

– `interpret`: it encapsulates the human knowledge specifying how high abstraction level inputs are produced from low abstraction level driver information or how high abstraction level command are performed by low abstraction level driver actions. This activity owns a dynamic and functional[3] behavior. The dynamic behavior repre-

2. expect for event where format() is not applicable
3. In SAIA, the functional behavior possess a temporal budget called WCET: Worst Case Execution Time

sents the reactive part of the interpretation (way to collect and produce information) whereas the functional behavior represents the transformation part of the interpretation (way to compute pieces of information together: average, max, ...).

– `qosAdapt`: it specifies a modification/adaptation of the QoS. This activity is mainly represented by the dynamic behaviors (creation of a constant delay, ...) but can also embedded a functional behavior for complex computation (i.e. interpolation).

The association of the three previous layers (SAM + ALM + SAIM) produces a representation of the whole system called SASM (Sensors Actuators Specific Model).

2.5. *SAIA and the QoS*

Since SAIA goal is the verification of the QoS conformance during conception, no online QoS information is needed. Thus, SAIA specifies the QoS in a lightweight way, like in (L.DiPippo *et al.*, 1999). The QoS characteristics are specified by interval [Min ; Max]. Once specified, SAIA mixes two different approaches for QoS consideration. In one hand, the SAIM specifies the required QoS and the SAM specifies the provided QoS. This results in a contract for the QoS in the sense of the fourth contract level described in (Beugnard *et al.*, 1999) for component contract specification. On the other hand, the QoS provided by the ALM is computed from the QoS provided by the SAM and the adaptation element behaviors. It is evaluated by the analysis of a specific configuration.

Now, the concepts and terms used by SAIA have been introduced. The next section describes the proposed method by beginning with the SAIM modeling.

3. SAIA MDE method

In SAIA the control activity is developed independently of the sensors actuators. In consequence, the development method starts with the modeling of the SAIM and its validation. Then, the SAM and the ALM modeling are presented. To finish, the models and the models transformations used for the validation are detailed. It is important to notice that all models and models transformations are done during the conception of the system. No transformations are realized on the deployed system.

3.1. *SAIM development and validation*

To begin, it is necessary to produce the SAIM from the specifications. The first stage is the determination of the I/O needed by the control. It is important to notice that the I/O represent data, event or command which are independent of any sensor or actuator technology. The I/O identification is extracted from the specification according to the functionality needed by the control activity (labeled 1 on figure 3). After the I/O identification, it is still impossible to describe the QoS required because it highly

depends on the control activity. Therefore the second stage is the control activity modeling (labeled 2 on figure 3). The control model must be based on the I/O description; the other information is extracted from the specifications. The modeling of the SDPC control activity is not treated here because SAIA sees it as a black box. The following stage (labeled 3 on the figure 3) extracted the non-functional constraints from the specifications. These constraints reflect the QoS objectives for the application in term of deadline, power consumption and user level quality of service (i.e. no deviation in the trajectory, etc). These constraints are a goal to reach and the constraints derivation explicitates that the QoS objectives are met as long as the I/O QoS required is respected (Torngren, 1998). This models transformation produces the QoS required for each I/O (labeled 4 on the figure 3). Now, the I/O functional model, the QoS required by the I/O and the control model can be mapped to produce a single model: the SAIM (labeled 5 on figure 3).

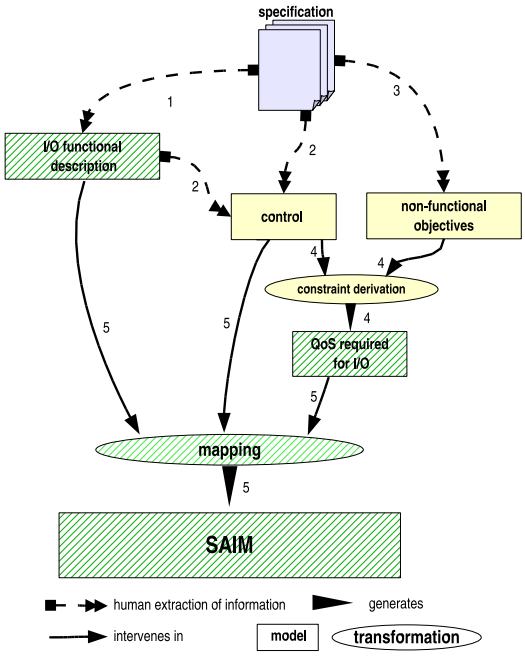


Figure 3. *SAIM development process*

3.2. *SAM modeling*

The second stage in the system development is to choose and model the sensors actuators of a specific platform. The output of this stage is the SAM. The SAM specifies the platform driver types and their provided QoS. Once the SAM modeled, we must

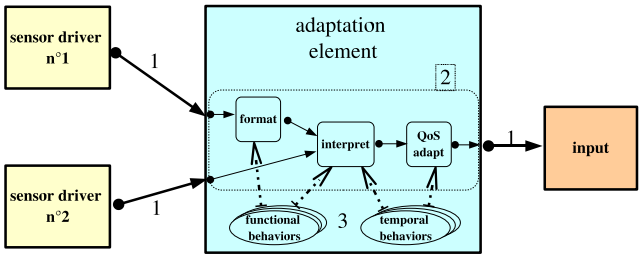


Figure 4. *ALM development process*

link the SAM with the I/O of the SAIM. It must be done by respecting the functional as well as extra-functional constraints.

3.3. *ALM modeling*

Since the ALM is in charge of the complex link between the SAM and the SAIM, its modeling specifies the mapping between these two models. The ALM modeling consists in three main stages. The first one (labeled 1 on the figure 4) introduces the structuring knowledge reflecting the drivers involved in the construction/accomplishment of the I/O. In other words, connectors are created between the drivers interfaces and the associated adaptation elements interfaces as well as between the adaptation elements interfaces and the associated I/O interfaces. This stage requires pieces of information from both the SAM and the SAIM (labeled 1 on the figure 5). The two other stages consist in the modeling of the internal part of each adaptation elements. In the second stage (labeled 2 on the figure 4), a structuring of the internal entities must be done (i.e. Is format useful? Is QoS adaptation useful and where, before or after the interpretation? ...). This structuring is mainly dictated by the domain knowledge and experiences. Then, in the last stage (labeled 3 on the figure 4), for each internal activity of the adaptation elements the dynamic and functional behaviors have to be specified. It gives information on the elements behaviors as described in the section 2.4 (interpretation type, functions WCET, and so on).

Once the ALM modeled, a mapping of the three models (SAM SAIM and ALM) generates the SASM (labeled 2 on the figure 5), i.e. a Sensors Actuators Specific Model of the whole system. At this stage, the SASM is functionally complete but the QoS provided by the ALM is not known. The remainder of the method is dedicated to the completion and the validation of this model.

3.4. *System validation*

The system validation is decomposed into two main stages. The first one consists in the evaluation of the QoS provided by the ALM. The second stage consists in a

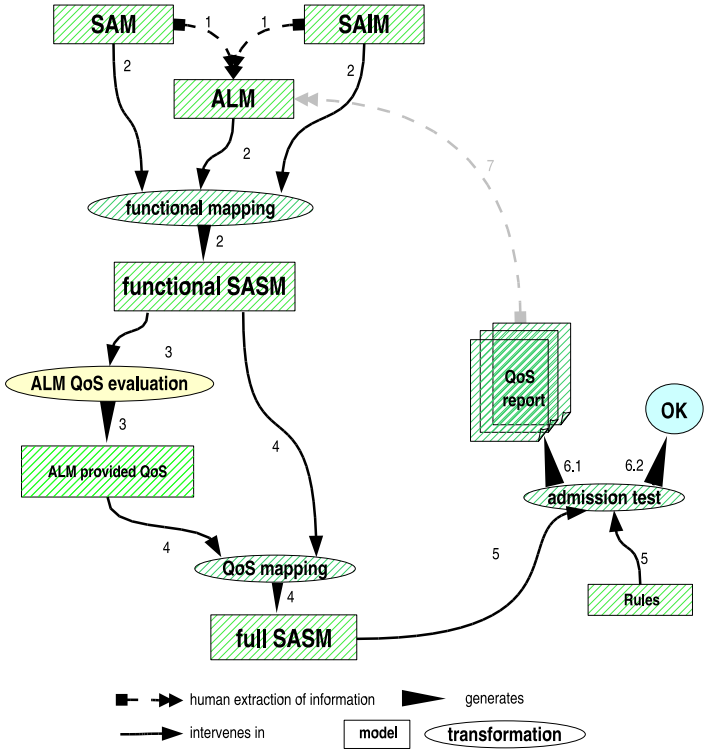


Figure 5. *SASM models and transformation for validation*

comparison between the QoS provided by the ALM and the QoS required by the SAIM. This second stage is called admission test.

3.4.1. *ALM provided QoS evaluation*

To produce a model of the ALM provided QoS, the following information must be available for all adaptation element in the ALM:

– the QoS provided by the sensor drivers (for Input production) or by the Output (for actuator driver),

– the adaptation element internal structure,

– the adaptation element activity temporal behaviors.

Now, all these peaces of information must be analyzed and computed to produce the adaptation element provided QoS. This step has to be done for each adaptation element in order to produce the ALM provided QoS (labeled 3 on the figure 5). When finished, the ALM provided QoS model is mapped with the SASM to generate a full

SASM; i.e a SASM including functional and extra-functional information (labeled 4 on the figure 5). The determination of its validity is in charge of the admission test.

3.4.2. *Admission test*

To be valid, the provided QoS characteristics of the ALM must satisfy the required QoS characteristics of the SAIM. In SAIA, the QoS characteristics are specified thanks to intervals [Min ; Max]. Consequently, the admission test must verify these rules:

$$\{\forall(QoScharacteristic) \in ALM\} \subseteq \{(associated\,QoScharacteristic) \in SAIM\}$$

Therefore, for each QoS characteristics:

$$[Min_{provided}; Max_{provided}] \subseteq [Min_{required}; Max_{required}]$$

The admission test transformation can generate two outputs depending on its success or its failure. If the admission test fails (labeled 6.1 on the figure 5), a QoS report is generated to identify where the rules violation have occurred. This report can be examined and the QoS adaptation of the responsible element(s) has to be done or changed (labeled 7 on the figure 5) and a new validation process must be done. If the admission test succeeds (labeled 6.2 on the figure 5), the validation process ends.

4. The SAIA method implementation

In the previous section, all the models and models transformations for a SAIA MDE development and validation have been presented. In order to implement the proposed method, a SAIA development environment has been developed. The generation of this specific modeling environment is based on the Model Integrated Computing (Sztipanovits *et al.*, 1997). MIC uses meta-models to explicit the rules governing the modeling of valid systems. MIC is implemented by the Generic Modeling Environment (GME (ISIS, 2005)), a meta-programmable toolkit for the creation of domain-specific modeling environments. The meta-modeling paradigm employed in GME is based on the UML.

Consequently, SAIA has been meta-modeled in GME meta-modeling paradigm. After the meta-modeling of the SAIA entities in GME, presentation information must be added to the meta-model for the generation of the SAIA modeling tool. This information describes various views where entities are viewable or not according to the model (see 1: in the figure 6) and the place in this model. Since each model is described in a different view, in a specific model, only the entities belonging to this model are usable (see 2 in the figure 6). In addition of the different views, it allows creating a depth hierarchy. For instance, an entity such as the `interpret` activity is only accessible when "entering" in an adaptation element (see 4 in the figure 6). This way, it separates the concerns between the different models but also between the structure of the system (see 3 in the figure 6), the internal structure of components (see 4

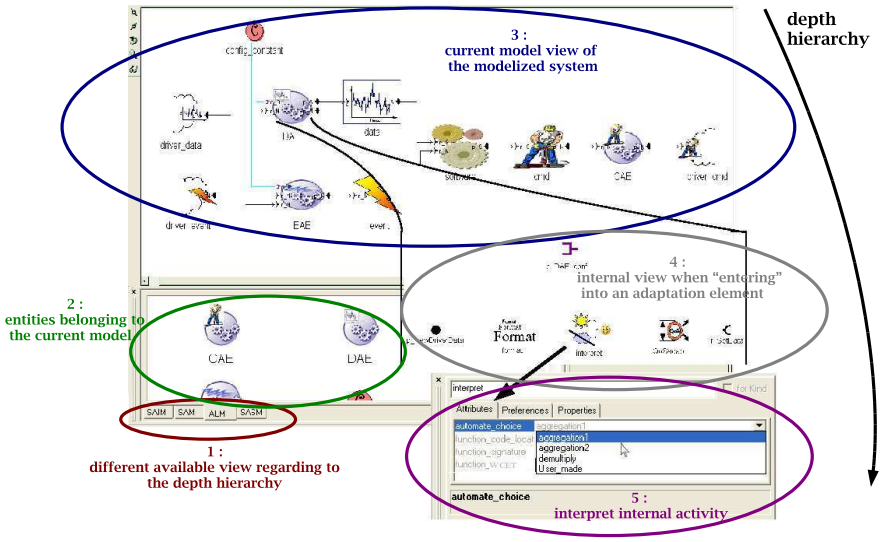


Figure 6. *a view of the tool and its hierarchy*

in the figure 6) and the behaviors of component internal activities (see 5 in the figure 6). A screenshot of the tool and its different views are given figure 6.

Now, it is possible to use GME with SAIA paradigm, i.e. it is possible to create most of models described by the method. On the figure 3 and 5, all the models and transformations represented by a hatched box are supported by the tool. The other models and models transformations includes the modeling of the control and the constraints derivation transformation. For SDPC, the control model and the constraint derivation can be implemented by the MATLAB tool suite (Mathworks Inc, 2005a, Mathworks Inc, 2005c, Mathworks Inc, 2005b).

The use of SAIA has highlighted the use of redundant interpretation and QoS adaptation reactive part. Consequently, a toolbox of predefined automata for `interpret` and `qosAdapt` activities is provided for SAIA. The interpretation, as well as the QoS adaptation, modifies the QoS provided by the SAM. To facilitate the evaluation of the new provided QoS, all automata in the toolbox are associated with equations. These equations characterize the QoS provided in output of the automaton according to the QoS provided in input and the automaton characteristics (DeAntoni *et al.*, 2005a).

This way, if the modeler uses only predefined automata, the transformation called: "ALM QoS evaluation" on the figure 5 can be done by applying the equations to the QoS characteristics provided in input of the automaton. On the contrary, if the modeler chooses user-made automata, analytical equations are not available and the validation is more complex. In this case, the tool acts as an input generator for analysis tools allowing the evaluation of the ALM provided QoS.

One solution has been implemented by the tool. It is based on timed automata modeling and analysis. The chosen language for the timed automata modeling is IF (Bozga *et al.*, 2002). IF is composed by a formal timed automata language and an associated tool. It allows the evaluation of some QoS characteristics (Ben-Hedia *et al.*, 2005). Consequently, the SASM described in the SAIA modeling tool is translated into an IF model. For the first stage of this translation, we use the faculty of GME to export models in an XML format whose the DTD is a representation of the modeling paradigm used. In the second stage of the translation, a parser retrieves the necessary information from the XML file. Then it generates the appropriate IF model from this information. Now, the analysis can be performed thanks to the analysis tools suite.

After these steps, the QoS provided by each adaptation element and accordingly by the adaptation layer is known and can be added to the SASM model in the tool. The admission test must now verify that the QoS provided by the adaptation layer satisfies the QoS required by the SAIM.

The rules governing the admission test are the ones described in the section 3.4. This admission test is implemented in the tool thanks to OCL constraints. The evaluation of these constraints can be done in the SAIA modeling tool by asking for constraints evaluation. If the test fails, for each constraint violations the tool gives the adaptation element and the QoS characteristic where the constraint is violated.

This tool has been used with success in the realization of an exploration robot (DeAntoni *et al.*, 2005c). This exploration robot must conform to the specification dictated by the maRTian Task challenge. After the modeling of the robot, the source code must be generated. This code generation needs information about the Real Time Operating System (available scheduling policy, etc) where the system is executed. It also needs mapping rules specifying how the components and their activities are mapped into RTOS tasks. Since the code generation goal was only to verify that all the mandatory pieces of information are included in the full SASM model, this knowledge has been brought manually.

The method advantages are highlight by the fact that the exploration robot development relies on a simulator. The SAM in this case represents the services provided by the simulator. Since the SAM is separated from the SAIM by the ALM, during the deployment on a real target, the changes are not propagated in the whole system. The changes impact the SAM and the ALM but the SAIM stays unchanged.

5. Related work

In SAIA and the proposed method, a clean separation has been done between the control activity and the sensors actuators provided by a specific target. This approach is to assimilate to the MDA (Model Driven Architecture: (OMG-MDA, 2003)) philosophy where a Platform Independent Model (PIM), a Platform Model (PM) and a Platform Specific Model (PSM) are defined. The proposed method is an implementation of the MDA philosophy where the platform is identified to the Sensors and Actuators.

MDA is successfully used for the development of general purpose systems (Frankel *et al.,* 2002, D.Wampler, 2003, P.Caceres *et al.,* 2003) but there are few MDA-based implementations for dedicated system where correctness strongly relies on hardware performance.

(P.Boulet *et al.,* 2003) proposes an implementation of MDA for systems on chip design. This MDA adaptation of the "Y-approach" introduces an association model in order to model the mapping between software and hardware models. This approach considers machines and buses as the PM but not sensors and actuators dependencies. Moreover, contrary to the proposed approach, no extra-functional descriptions are made. However, the two approaches use the same idea which is the use of an intermediate model to link the platform specific and platform independent models.

In the software architecture domain, lots of different works and strategies have already been proposed for the development and the validation of systems. One approach is specifically closed to the one described in this paper: the approach developed for the REACT project (Faucou *et al.,* 2004).

The REACT project is based on the architecture description language CLARA (Durand, 1998), dedicated to the description of reactive systems architecture. The first stage of the project concerns the validation of a candidate architecture regarding its functional and extra-functional requirements. The similar point with the proposed approach is the use of external formalisms and tools to validate the systems. However, conversely to the proposed approach, the CLARA description does not identify various models and models tranformations to conduct the architecture building.

Sensors / actuators communication is seldom dealt with but, (Cristina *et al.,* 2005) proposes a component based package for modeling of sensors and actuators in an OSGI context: SensorBean. The approach identifies some services which are also identified in SAIA. Since it is a recent approach, there is no detailed information about extra-functional requirements, specification, and validation.

6. Conclusion

This paper presents a model driven methodology for the development and validation of SDPC communication protocol. Various models and model transformations are identified. It facilitates the management of the SAIA entities and constraints during development and validation of systems. The generation of a tool supporting the method allows a systematic used of the models and transformations. Moreover, it has allowed the validation of the approach through an international challenge: maRTian task (DeAntoni *et al.,* 2005c).

Several orientations for the future works can be interesting. First, for an use in an industrial context, a SPEM (OMG-SPEM, 2005) modelisation of the proposed process could facilitate evolution and reuse through systems families. Another interesting orientation is to allow the creation of a bridge between the SAIA modeling tool and

UML modeling/simulation tools; i.e. the creation of a SAIA UML profile. This way a transformation from the SAIA modeling environment to a UML editor could be implemented. Next orientation is the formalization of the transformation between SAIA and IF models at a meta-model level. Finally, SAIA could be extended in order to allow an automatic code generation for RTOS.

7. References

Bass L., Clements P., Kazman R., « Software Architecture in Practice », *Addison Wesley, Reading, Mass*, 1998.

Ben-Hedia B., Jumel F., Babau J.-P., « Formal evaluation of Quality of Service for data acquisition systems », *Forum on specification and Design Language*, 2005.

Beugnard A., Jézéquel J.-M., Plouzeau N., Watkins D., « Making component contract aware », *IEEE computer, 32(7)*p. 38-45, 1999.

Bozga M., Graf S., Mounier L., « IF-2.0: A validation environment for Component-Based Real Time systems », *In ed. Brinksma, K.G. Larsen (Eds) Proceedings of CAV'02*, 2002.

Cristina M., Donsez D., Lalanda P., « Approche IDM pour le développement des services basés capteurs », *Ingénierie Dirigée par les modèles (IDM'05)*, 2005.

DeAntoni J., Babau J.-P., « A MDA approach for systems dedicated to process control », *eleventh IEEE International Embedded and Real Time Computing Systems and Applications (RTCSA'05)*, 2005a.

DeAntoni J., Babau J.-P., « A MDA-based approach for real time embedded systems simulation », *Nineth IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'05)*, 2005b.

DeAntoni J., Babau J.-P., « SAIA: Sensors/actuators Independent Architecture - A showcase through maRTian task specification », *Proceedings of the ERTSI 2005 - Embedded Real Time Systems Implementation Workshop, held in conjunction with 26th IEEE International Real-Time Systems Symposium*p. 43-50, 2005c. http://www.cs.york.ac.uk/ftpdir/reports/YCS-2005-397.pdf.

Durand E., « Description et vérification d'architectures d'applications temps réel: CLARA et les réseaux de Petri temporels », *PhD thesis, école centrale de Nantes*, 1998.

D.Wampler, « The Role of Aspect-Oriented Programming in OMG's Model-Driven Architecture », *Aspect Programming Inc*, 2003.

Faucou S., Déplanche A.-M., Trinquet Y., « An ADL centric approach for the formal design of real time systems », *In Architecture description language, IFIP*p. 67-82, 2004.

Frankel D., J.Parodi, « Using Model-Driven Architecture(tm) to Develop Web Services », *IONA Technologies PLC, White paper*, 2002.

Ghezzi C., Jazayeri M., Mandrioli D., « Software Architecture: a Roadmap », *In A. Finkelstein editeur, International Conference on Software Engineering, ACM press*, 2000.

ISIS, « The Generic Modeling Environment (GME) », 2005. ISIS: Institute for Software Integrated Systems. Vanderbilt University.

Jumel F., « Definition and management of a quality of service for real time applications *(in french)* », *thesis in LORIA laboratory, Nancy*, 2003.

L.DiPippo, L.Ma, « A UML Package for Specifying Real-Time Objects », *Computer Standards and Interfaces 2000*, 1999.

Mathworks Inc, « MATLAB », *http://www.mathworks.com/products/matlab/*, 2005a.

Mathworks Inc, « real time workshop », *http://www.mathworks.com/products/rtw/*, 2005b.

Mathworks Inc, « SIMULINK », *http://www.mathworks.com/products/simulink/*, 2005c.

OMG-MDA, « Model Driven Architecture guide V1.0.1 », *http://www.omg.org/mda*, 2003.

OMG-SPEM, « Software Process Engineering Metamodel Specification Version 1.1 », *http://www.omg.org/docs/formal/05-01-06.pdf*, 2005.

P.Boulet, J.L.Dekeyser, C.Dumoulin, P.Marquet, « MDA for SoC Embedded Systems Design, Intensive Signal Processing Experiment », *FDL03*, 2003.

P.Caceres, E.Marcos, B.Vela, « A MDA-Based Approach for Web Information System Development », *wisme*, 2003.

Robles E., Held J., « A comparison of Windows driver model latency performance on Windows NT and Windows 98 », *Proc. OSDI third symposium*, 1999.

Shaw M., Garlan D., « Software architecture: Perspectives on an emerging discipline », *Prentice Hall*, 1996.

Sztipanovits J., Karsai G., « Model-Integrated Computing », *IEEE Computer, vol. 30, no. 4, pp. 110-112*, 1997.

Torngren M., « Fundamentals of implementing Real-Time Control applications in distributed computer systems », *Real Time System*, 1998.

Processus d'imitation pour patrons de conception à variantes

Nicolas Arnaud — Agnès Front — Dominique Rieu

LSR-IMAG, équipe SIGMA
681 rue de la passerelle
BP 72 38402 Saint Martin d'Hères Cedex
{prenom.nom}@imag.fr

RÉSUMÉ. *La réutilisation des patrons de conception ne peut être réduite à une simple adaptation de la solution au contexte. En effet, les patrons comportent des spécifications génériques, répondant bien souvent à des descriptions variables. Afin de garantir une réutilisation correcte et traçable, nous proposons l'emploi d'un processus dit d' « imitation » où l'ingénieur de patrons spécifie tout d'abord le modèle imitable (à variantes et générique) de sa solution. Par la suite, le concepteur de systèmes d'information va être guidé à travers plusieurs activités lui permettant, dans un premier temps, de choisir les variantes qu'il désire imiter pour ensuite appliquer ce choix au contexte d'imitation. Cet article présente une manière de spécifier des solutions complètes, à variantes et génériques, en utilisant et étendant UML et en donne les implications au sein du processus d'imitation.*

ABSTRACT. *Design patterns reusability is more than a simple adaptation of the solution, regarding to the context. Patterns allow generic specification linked to variable descriptions. We purpose an "imitation" process in order to vouch a correct and traceable reuse where, at first, the pattern engineer specifies an imitable model containing both variant and generic aspects. Afterwards, the information system engineer will be driven through some activities first allowing him to choose variants he wants to use, and then follow the appliance of his choice to the imitation context. This paper presents a way of specifying complete, variable and generic solutions using and extending UML. We also present how to take advantage of these specifications during the imitation process.*

MOTS-CLÉS : *patrons de conception, variabilité, processus d'imitation, UML 2.*

KEYWORDS : *design patterns, variability, imitation process, UML2.*

1. Introduction

L'exigence de qualité des systèmes d'information implique rigueur et continuité dans les différentes phases de développement. Il est donc crucial de capitaliser les savoirs et les savoir-faire afin de les réutiliser au cours d'autres processus de développement. La réutilisation est généralement considérée comme un gage de cette qualité, et particulièrement si elle met en œuvre une traçabilité modulaire des spécifications. Les patrons (Alexander, 1979), appliqués à l'ingénierie des systèmes, sont des outils particulièrement pertinents pour ce type d'approche.

Un patron décrit un problème fréquemment rencontré dans un contexte ainsi que la solution consensuelle qui le résout. Dans le domaine des systèmes informatiques et pour ce qui nous intéresse dans celui de la conception de systèmes d'information, on peut bien évidemment citer les patrons de conception et parmi les plus connus ceux du Gang of Four (GoF) (Gamma et al., 1995), qui nous serviront à la fois de référence et d'exemple.

Bien que complétée par de nombreuses informations, la solution d'un patron est souvent limitée à un diagramme de classes UML (OMG, 2005), qui est pour nous une spécification incomplète de cette solution. Nous proposons de spécifier les patrons sous la forme d'un mini-système à trois vues : fonctionnelle (cas d'utilisation), dynamique (diagrammes de séquences) et statique (diagrammes de classes).

De plus, dans de nombreux cas, et particulièrement pour le GoF, la description du patron est clairsemée d'informations exprimant des variantes possibles de cette solution, et ceci selon tous les aspects : fonctionnels, dynamiques ou statiques. Nous intégrons cette variabilité au sein de la spécification des patrons afin d'instrumenter la sélection de variantes dans le processus de réutilisation. Ce dernier, appelé processus d'imitation, permet d'extraire la solution du patron et de l'appliquer dans un système en construction. Cet article décrit ce processus ainsi que ses produits et activités.

Cependant, variabilité et complétude ne sont pas les seuls besoins lorsqu'il s'agit d'appliquer un patron de conception. L'adaptation au contexte est une notion très importante que nous désirons également mettre en œuvre dans notre processus d'imitation. La généricité des solutions proposées par l'ingénieur de patrons doit elle aussi être instrumentée dans notre processus. Le modèle complet, variable et générique de la solution est appelé le modèle imitable car c'est lui que le concepteur d'application veut utiliser lors de la construction de son système d'information.

La figure 1 illustre le processus d'imitation qui est composé de deux sous-processus : le processus de réduction et le processus d'application. Le premier permet à l'ingénieur d'applications de sélectionner le patron à réutiliser puis, à partir du modèle imitable de ce patron, de sélectionner les variantes du patron qu'il désire imiter. Le processus de réduction rend un modèle imité dans l'état adaptable, réduit aux variantes sélectionnées. Le processus d'application réalise l'adaptation du

modèle imité [adaptable] ainsi que son intégration au modèle du système d'information en construction.

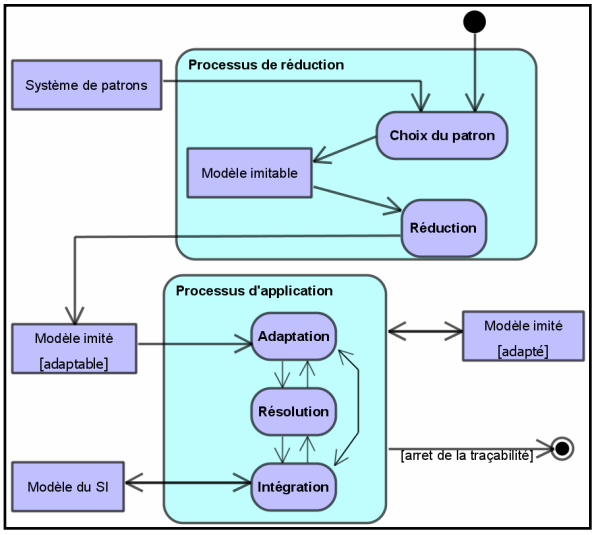


Figure 1. *Processus d'imitation*

Dans la section 2, nous décrivons les concepts de variabilité et de généricité mis en œuvre dans le mini-système à 3 vues composant le modèle imitable et proposons des extensions du méta modèle UML destinées à la spécification de ce dernier. Le processus de réduction et le processus d'application sont respectivement détaillés et illustrés dans les sections 3 et 4. Le patron « Observateur » du GoF sert d'exemple et d'illustration tout au long de l'article.

2. Modèle imitable

2.1. Les concepts

Complétude des spécifications. Le domaine de la conception des systèmes d'information est riche de processus de développement (RUP, 2TUP,…) qui combinent aspects fonctionnels, dynamiques et statiques. Nous proposons une approche semblable pour rendre les solutions de patrons plus complètes qu'avec un simple diagramme de classes UML et considérons que l'imitation de telles solutions apportera une meilleure qualité de réutilisation. Nous proposons ainsi de représenter la solution d'un patron sous la forme d'un mini-système composé de 3 vues :

− la vue fonctionnelle qui présente sous forme d'un diagramme de cas d'utilisation, les fonctionnalités du système en construction, les dépendances qui les relient et les acteurs qui les déclenchent ;

− la vue dynamique qui présente sous forme d'un diagramme de séquence, les collaborations entre les différentes classes introduites dans la solution ;

− la vue statique qui exprime, sous forme d'un diagramme de classes, la structure des entités du système ainsi que la réalisation de leurs relations.

Variabilité. Indépendamment de cet aspect « mini-système à 3 vues », certaines informations variables doivent être prises en compte dans la solution d'un patron, par exemple le fait que les fonctionnalités offertes par la solution d'un patron soient essentielles ou facultatives. Nous proposons d'utiliser le mécanisme de **point de variation** pour représenter les endroits du système où il y a une variation (Czarnecki *et al.*, 2000), c'est-à-dire où des choix vont devoir être faits afin d'identifier les **variantes** à utiliser. Selon (Bachmann *et al.*, 2001), il existe plusieurs types de variabilité pour un point de variation :

− les options : choix de zéro ou plusieurs variantes parmi plusieurs,

− les alternatives : choix d'une variante parmi plusieurs,

− les alternatives optionnelles : choix de zéro ou une variante parmi plusieurs,

− les ensembles d'alternatives : choix d'au moins une variante parmi plusieurs.

Généricité. Certaines propriétés génériques non fonctionnelles doivent finalement pouvoir être exprimées par l'ingénieur de patrons. Il s'agit par exemple, dans le patron « Observateur » de considérer que la classe *Concrete_Observer* peut être dupliquée au sein d'une même imitation. C'est d'ailleurs le cas dans l'exemple de la rubrique *Motivation* décrite par le GoF. Ce genre de propriétés est pris en compte lors de l'application du modèle imité [adaptable] au contexte (cf. section 4). La généricité ne peut cependant s'exprimer qu'au niveau de la vue statique de la solution.

Nous rappelons ici notre objectif de traçabilité du processus d'imitation, qui implique que le concepteur d'applications ne peut pas faire ce qu'il veut de son modèle imité et qu'il doit se soumettre aux contraintes de généricité fixées par l'ingénieur de patrons. L'instrumentation de ce processus d'imitation implique donc la mise en œuvre d'un mécanisme de prévention et de vérification des imitations.

2.2. La vue fonctionnelle à variantes

2.2.1 Problématique

Un modèle de cas d'utilisation présente les fonctionnalités du système en construction, les dépendances qui les relient et les acteurs qui les déclenchent. De ce résultat issu de l'étude des besoins dépend le reste de la spécification. Si la solution

d'un patron apporte plusieurs fonctionnalités, nous proposons de les faire apparaître explicitement.

Nous rappelons l'intention du patron «Observateur» : « *Définir une dépendance entre les observateurs d'un même sujet telle que, quand le sujet change d'état, tous ces observateurs soient informés et mis à jour* » (Gamma et al., 1995). On peut donc déduire deux fonctionnalités : modifier le sujet (*subject modification*) et gérer les observateurs (*observers management*). La première consiste en la modification du sujet avec mise à jour des observateurs, la seconde traite de l'ajout et de la suppression des observateurs au sujet.

Ainsi, le patron « Observateur » ne se borne pas à mettre en œuvre la notification des observateurs, mais permet également d'attacher ou de détacher ces derniers à un sujet. Cette fonctionnalité certes pratique n'est pas indispensable. La solution doit donc introduire le fait que la gestion des observateurs ne constitue qu'une fonction secondaire qui peut ne pas être retenue lors de l'imitation.

De même, si la fonctionnalité principale du patron « Observateur » est de permettre la modification de l'état du sujet avec une mise à jour automatique et donc implicite des observateurs, il est également possible que le déclenchement de cette mise à jour soit à la charge exclusive de celui qui initie le changement d'état. Les deux cas sont discutés dans la rubrique *Implémentation* du patron.

2.2.2 Solution proposée

La spécification du mini-système doit prendre en compte non seulement les aspects variables au niveau des fonctionnalités, mais également leurs conséquences sur les vues dynamique et statique. Dans (Ziadi *et al.*, 2005), les stéréotypes <<variation>> et <<variant>>, représentant respectivement un point de variation et une variante, sont définis pour exprimer la variabilité sur les fragments d'interaction des diagrammes de séquence UML2.

Les deux notions principales que sont les points de variation et les variantes sont indispensables à la spécification d'un modèle imitable, c'est pourquoi nous les intégrons à UML2 par l'ajout de deux nouvelles méta-classes *VariationPoint* et *Variant,* toutes deux sous-classes de *VariationElement*. Une variante n'est variante que d'un seul point de variation et un point de variation possède au moins une variante. Tout élément de variation (*VariationElement*) possède un nom.

Nous n'introduisons, dans cet article, que deux relations de variabilité (*VariationRelationship*) entre cas d'utilisation : l'alternative (*Alternative*) et l'option (*Option*). Cette approche diffère de celle de (Van der Maβen et al., 2002) qui considère les options comme des points d'extension particuliers.

Pour toute relation de variabilité, il existe un point de variation (*VariationPoint*) représenté par un cas d'utilisation (*base*) stéréotypé <<variation>> dans la vue fonctionnelle, portant le même nom. Le nombre de cas d'utilisation <<variant>>

dépend alors du type de variation (ici option ou alternative). Là aussi, les cas d'utilisation portent le même nom que les variantes.

Un cas d'utilisation est attaché au client par le biais d'une *Association*. Dans ce cas c'est une fonctionnalité du patron de premier niveau, c'est-à-dire qu'elle peut être primaire ou secondaire. Nous définissons les deux stéréotypes de *Association* suivants :

− <<primary>> pour caractériser qu'un cas d'utilisation est primaire, tel *subject modification*.

− <<secondary>> pour caractériser qu'un cas d'utilisation est secondaire, tel *observers management*.

La figure 2 illustre ces extensions du méta-modèle. Les méta-classes ajoutées sont présentées en foncé.

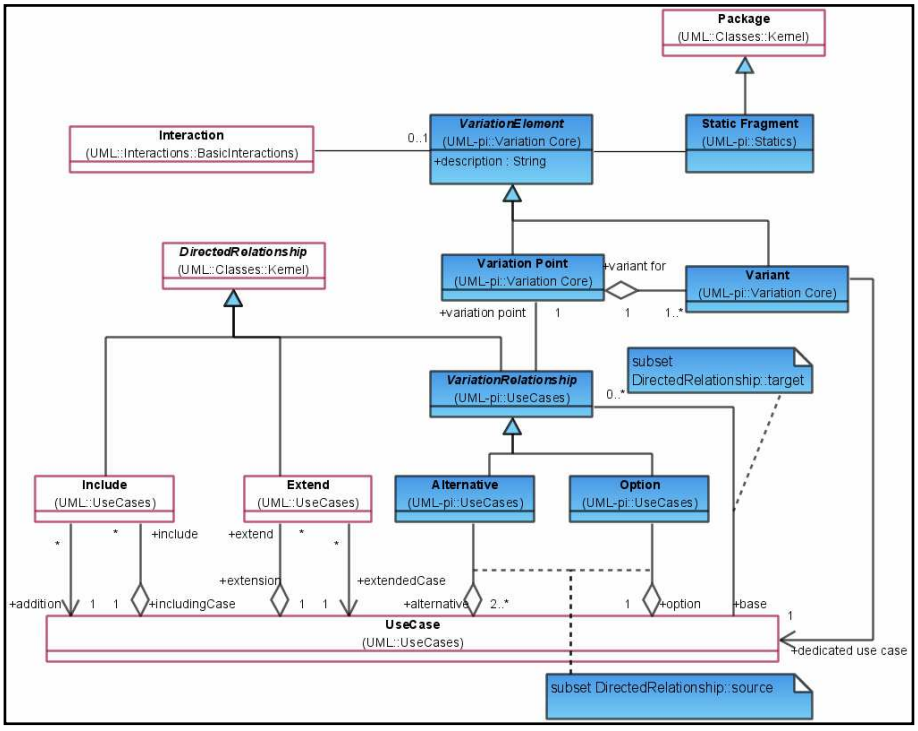


Figure 2. *Partie « variabilité » du méta modèle pour les modèles imitables.*

La figure 3 illustre l'utilisation de ce méta-modèle pour spécifier la vue fonctionnelle de « Observateur ». On y retrouve les alternatives de notification explicite (*explicit notification*) et implicite (*implicit notification*). L'acteur de la vue des cas d'utilisation correspond au « client » des patrons du GoF. Ce dernier spécifie

les points d'entrée qu'il utilise pour accéder à la solution. L'acteur client est donc l'entité qui déclenche les fonctionnalités.

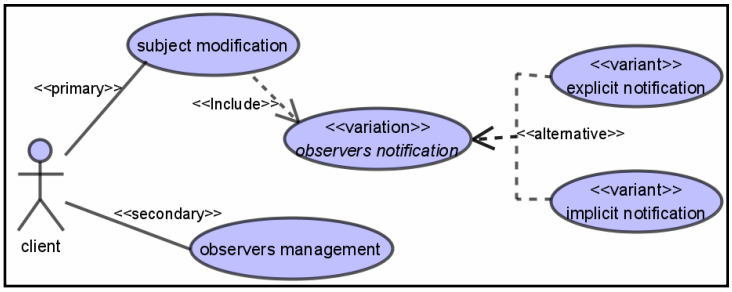


Figure 3. *Observateur : vue fonctionnelle à variantes*

Il est important de noter que les dépendances introduites (option, alternatives,…) sont, contrairement à l'inclusion ou à l'extension, utilisées lors de la réduction ; pendant la spécification du système d'information.

2.3. La vue dynamique à variantes

Dans la rubrique *Collaborations*, le GoF propose un diagramme de séquence décrivant le cas d'utilisation que nous avons appelé *subject modification*. Dans ce diagramme, c'est un observateur qui modifie le sujet mais, à priori, n'importe quel objet pourrait en faire de même, y compris le client. Il est donc nécessaire « d'étendre » ce diagramme de séquence afin de le rendre plus complet. Nous complétons la solution originale avec la méthode *updateSubjectState* afin de représenter le changement d'état du sujet (cf. figure 4). Ce que réalise précisément cette méthode n'est pas une préoccupation de ce patron. Le concepteur qui imitera ce patron aura à sa charge de définir cette méthode mais ne pourra remettre en cause son emplacement chronologique dans la séquence *setState*. Nous adoptons la même approche en ce qui concerne l'état de l'observateur, mis à jour dans la méthode *update* en utilisant sa méthode privée *setObserverState*.

En ce qui concerne la variabilité dynamique impliquée par la vue fonctionnelle, nous utilisons une approche similaire à (Ziadi *et al.*, 2005), en attachant à chaque élément de variabilité (point de variation ou variante) une *Interaction* qui est une « *unité de comportement qui se concentre sur un échange d'information observable entre deux éléments* » (OMG, 2005). C'est au sein d'une interaction que les envois de messages et autres fragments combinés sont utilisés.

La figure 4 illustre la vue dynamique à variantes du cas d'utilisation *subject modification* de la vue fonctionnelle de « Observateur ».

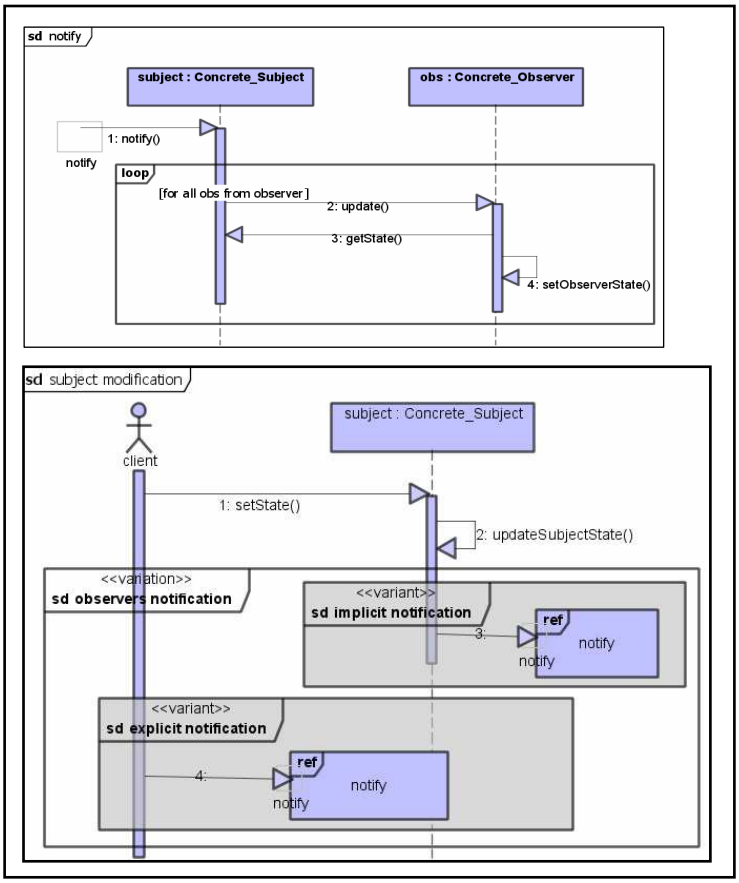


Figure 4. *Diagramme de séquence à variantes de subject notification*

2.4. La vue statique à variantes

Les diagrammes de classes expriment la structure des entités du système ainsi que la réalisation de leurs relations. Cette structure est en partie déduite de la vue dynamique, mais le concepteur y précise des propriétés statiques, par exemple les cardinalités des associations.

2.4.1 Problématique

La plupart des solutions de patrons se résument à une vue statique, en général un diagramme de classes. Ces structures sont souvent accompagnées de notes textuelles qui apportent quelques précisions. Cela peut aller du simple commentaire à l'algorithme. Dans le cas d' « Observateur », l'algorithme de la méthode *notify* est décrit à l'aide d'un pseudo-code. En considérant les diagrammes de séquence de la

vue dynamique, nous proposons de représenter la solution du patron d'une manière différente du diagramme de classes proposé par le GoF. Les principales différences avec la solution originelle du GoF sont les suivantes.

– Dans notre approche à mini-système, un algorithme est représenté dans la vue dynamique. C'est le cas pour la méthode *notify* et sa note explicative devient donc redondante.

– Nous ne matérialisons pas l'état du sujet concret car il peut être représenté de toute autre manière que par un seul attribut : plusieurs attributs, des liens avec d'autres classes, une combinaison des deux, etc. Cependant nous nous devons d'informer le concepteur d'applications du fait qu'il devra mettre en œuvre la représentation de cet état lors de son imitation. Nous proposons d'utiliser une note de type « à faire » (*TODO*).

– Nous avons ajouté à la vue dynamique une méthode privée de modification de cet état dont le contenu dépendra de la représentation choisie lors de l'imitation. C'est pourquoi la note « à faire » concerne l'état du sujet et sa modification.

Notons cependant qu'utiliser notre approche ne rend pas les solutions originelles caduques. Ces dernières restent un excellent support didactique et facilitent d'autant la compréhension du patron, c'est pourquoi il faut les conserver.

2.4.2 Variabilité statique

Il ne reste plus qu'à préciser les apports statiques de chaque fonctionnalité pour finaliser notre système. A ce niveau, notre proposition va s'écarter de la construction classique d'un système. En effet, nous suggérons de donner, pour chaque élément de variation, ses apports statiques au modèle de la solution. Ceci demande un effort de discrimination qui se révélera bénéfique au moment de la réduction.

La structure est disséminée dans plusieurs fragments qui s'assembleront pour former une vue statique classique lors de l'imitation. Toute classe « impactée » par une variation sera représentée dans le fragment statique (*Static* Fragment) de ce cas d'utilisation avec ses propriétés apportées. Ce dernier ne comportera que des informations structurelles (classes, associations, cardinalités, généralisation, etc…) du cas d'utilisation considéré. La plupart de ces propriétés statiques peuvent être déduites de la vue dynamique. La figure 5 présente les fragments statiques des cas d'utilisation *subject modification*, *observers notification*, *explicit notification*, *implicit notification* et *observer management*.

La visibilité de la méthode de notification n'est pas la même selon la variante de notification choisie. En effet, pour une notification explicite, cette méthode doit être publique pour pouvoir être déclenchée de l'extérieur. Ceci est d'ailleurs garanti par le message du diagramme de séquence lui-même. Mais, dans le cas d'une notification implicite, une visibilité protégée est plus adéquate. Dans la figure 5, la visibilité de la méthode *notify* n'est précisée qu'au niveau des variantes.

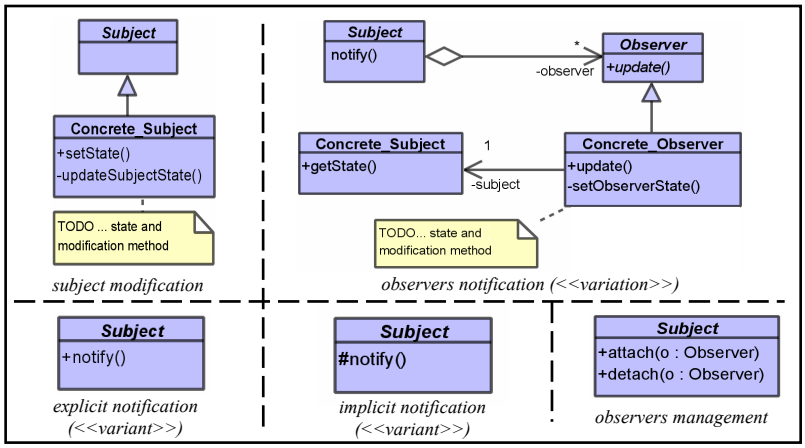


Figure 5. *Observateur : fragments statiques*

2.4.3 Généricité statique

Pour mettre en œuvre l'expression de la généricité et la garantie de sa juste utilisation, nous proposons d'étendre le méta-modèle d'expression de la structure statique utilisé dans les fragments statiques. Ainsi la méta-classe *Class* se voit étendue par une nouvelle méta-classe *ImitableClass* qui comporte un méta-attribut booléen supplémentaire : *duplicable*. Ce dernier permet à l'ingénieur de patrons de dire si, dans un fragment statique, une classe est duplicable lors de l'imitation. Toute métaclasse *MetaClasse* nécessitant l'expression de nouvelles méta-propriétés de généricité sera étendue par le même mécanisme dans la méta-classe *MetaClasseImitable*.

Une fois l'adaptation réalisée, il n'est plus nécessaire de conserver ce type d'informations dans le modèle imité [adapté], mais il faut néanmoins conserver un lien entre la classe adaptée et sa « source » imitable. C'est pourquoi nous définissons, pour chaque *MétaClasseImitable* une *MétaClasseAdaptée* permettant la traçabilité de l'adaptation. Cette approche est directement issue de nos précédents travaux (Arnaud et al., 2004). La figure 6 illustre cette extension du méta-modèle pour les classes.

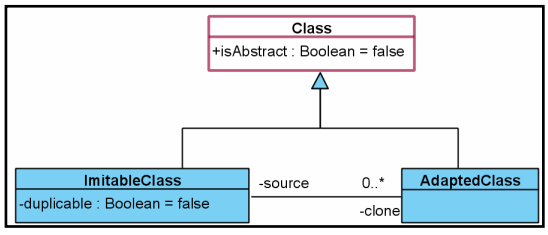


Figure 6. *Classe imitable et adaptée*

2.4.4 Cohérence inter-vues

Afin de garantir la cohérence entre les vues, il est nécessaire donner quelques règles, ici en langage naturel, mais qui pourraient s'exprimer à l'aide d'OCL :

– Un cas d'utilisation donne lieu à un fragment d'interaction (au moins) et un fragment statique de mêmes noms.

– Un cas d'utilisation de type « variation » implique un fragment d'interaction de type « variation ».

– Un cas d'utilisation de type « variant » implique un fragment d'interaction de type « variant » inclus dans le fragment « variation ».

– …

3. Processus de réduction

Le processus de réduction est constitué de deux activités. Il a pour but de transformer un modèle imitable en un modèle imité dans l'état adaptable.

– Le *choix du patron* à imiter consiste à sélectionner un patron dans un système de patrons. La solution du patron sélectionné est appelée un modèle imitable et consiste en un mini-système à variantes composé de trois vues. Cette activité n'est pas spécifique aux concepts que nous présentons dans cet article, mais aborde une problématique beaucoup plus générale, aussi nous ne la détaillons plus ici.

– La *réduction* permet au concepteur de choisir les variantes qu'il désire imiter à partir de la vue des cas d'utilisation. Ce travail est réalisé depuis la vue fonctionnelle du modèle imitable. Il consiste à sélectionner les variantes (et non les points de variations) et les fonctionnalités secondaires que l'on désire imiter, en respectant, le cas échéant, les cardinalités du type de variation (par exemple 1 parmi n pour une alternative).

A partir de cette sélection, il est possible de construire un modèle du mini-système épuré de toute information de variabilité fonctionnelle toujours composé de trois vues. Ce modèle est appelé le modèle imité mais il est pour l'instant dans l'état adaptable car il n'a pas encore été adapté au contexte. Ceci aura lieu dans le *Processus d'application*.

Construire la vue fonctionnelle du modèle imité [adaptable] est relativement aisé. Les cas d'utilisation sélectionnés sont gardés, les autres éliminés. Dans le cas d'une variante, cette dernière vient tout simplement remplacer son point de variation. Dans le cas d'une sélection multiple de variantes, les relations auxquelles participait le point de variation concerné sont dupliquées. Toute variante non sélectionnée est éliminée. Enfin, les stéréotypes de variabilité sont supprimés. L'acteur « client » est conservé. La figure 7 illustre la réduction fonctionnelle du patron « Observateur » via la sélection de l'alternative *implicit notification*. La partie droite représente donc la vue fonctionnelle du mini-système imité-adaptable.

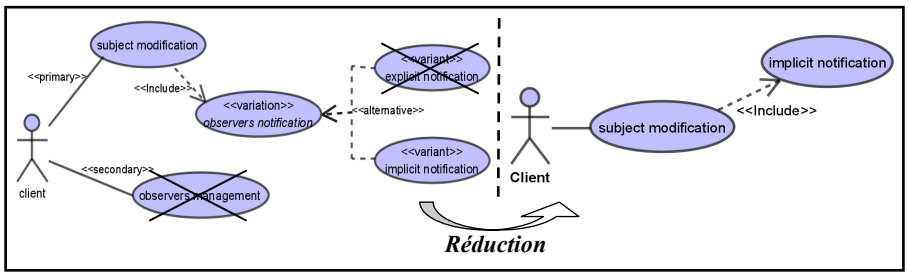


Figure 7. *Construction d'un modèle imité [adaptable] pour "Observateur"*

Les vues dynamique et statique sont alors automatiquement déduites. Dans la vue dynamique, tous les fragments ne correspondant pas à un cas d'utilisation sélectionné n'ont plus de sens et sont à ignorer. Les stéréotypes sont également supprimés. En ce qui concerne la vue statique il y a fusion des fragments correspondants aux cas d'utilisation sélectionnés. Nous ne nous attardons pas sur ces opérations puisqu'un patron bien spécifié ne devrait pas engendrer de problèmes de conflits. Les notes *TODO* et les propriétés génériques sont conservées jusqu'au processus d'application (cf. section 4). Afin d'assurer la traçabilité, chaque élément du modèle imité conserve une référence vers sa « version à variante ».

Les vues dynamique et statique issues de la réduction de la figure 7, sont présentées en figure 8 et figure 9. L'interaction de *notify* est identique à celle du patron imitable, c'est pourquoi nous ne la présentons pas de nouveau.

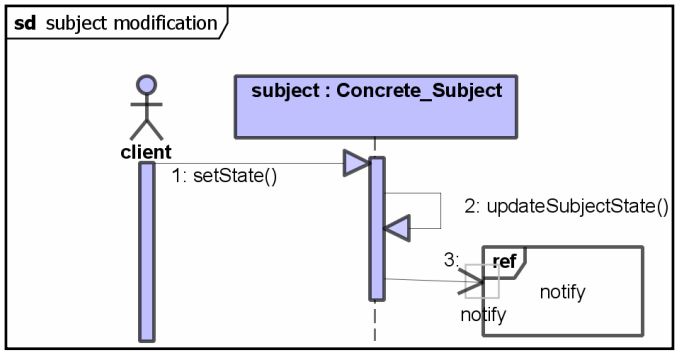


Figure 8. *Une réduction de la vue dynamique de "subject modification"*

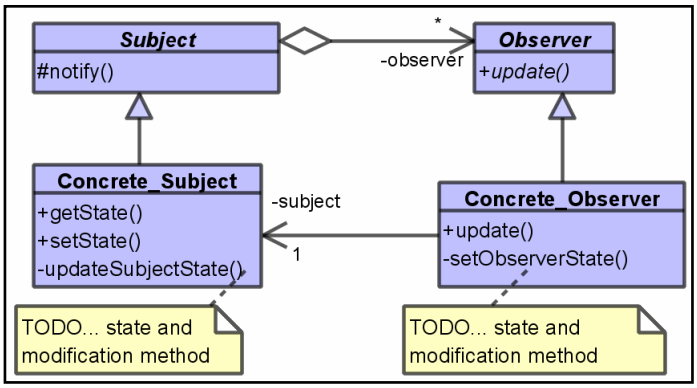


Figure 9. *Une réduction de la vue statique de "Observateur"*

4. Processus d'application

Ce processus est constitué de trois activités qui peuvent être exécutées de manière itérative jusqu'à obtenir un modèle imité dans l'état adapté, et donc intégré dans la spécification du système d'information.

– L'*adaptation* permet au concepteur de renommer les propriétés et de faire évoluer le modèle imité en restant conforme aux règles de généricité. Il peut par exemple dupliquer la classe d'observateur concret (cf. 2.4).

– La *résolution* concerne les notes *TODO*. Un modèle adaptable ne passera dans l'état adapté que si toutes les notes *TODO* sont résolues (cf. 2.4).

– L'*intégration* consiste à fusionner la spécification du modèle imité dans celle du système d'information. Un système est donc perçu comme un ensemble de modèle imité mais également de spécifications originales ne provenant pas d'une imitation de patron mais du savoir-faire du concepteur. (Arnaud et al., 2005) présente deux opérateurs pour l'intégration des imitations, aussi nous ne présentons pas plus cette activité dans cet article.

Nous détaillons ci-dessous les deux activités Adaptation et Résolution.

En sortie du processus de réduction, le concepteur adapte la spécification de la solution dans une forme prenant en compte le contexte d'imitation. Par exemple, si les observateurs sont, comme dans la *Motivation* donnée par le GoF, des diagrammes (histogramme, secteurs,…), la classe *Observer* peut être renommée *Diagram*.

Mais l'adaptation ne se borne pas à un renommage des éléments du modèle. Le mini-système imité [adaptable] est une solution sans équivoque, aux propriétés génériques près. Ainsi, la classe *Concrete_Observer* peut être dupliquée en plusieurs « exemplaires » (méta-propriété *duplicable* = true). La figure 10 présente une

adaptation de la vue statique du modèle imité [adaptable] précédent (cf. section 3) où les observateurs sont des diagrammes de deux sortes : histogramme (*HistoDiag*) et à secteurs (*SectorDiag*). Le sujet observé est une répartition (*Distribution*) des vents.

Le concepteur effectue également des actions de *résolution* consistant à résoudre les notes *TODO*. Dans la figure 10, la résolution de ces notes a été partiellement réalisée par l'insertion des attributs *north*, *south*, *east* et *west*. Le concepteur doit également spécifier le contenu de *updateValues*. Les résolutions des notes *TODO* peuvent être réalisées à tout moment. Certaines résolutions n'auront lieu qu'après l'intégration du modèle imité à la spécification du système d'information.

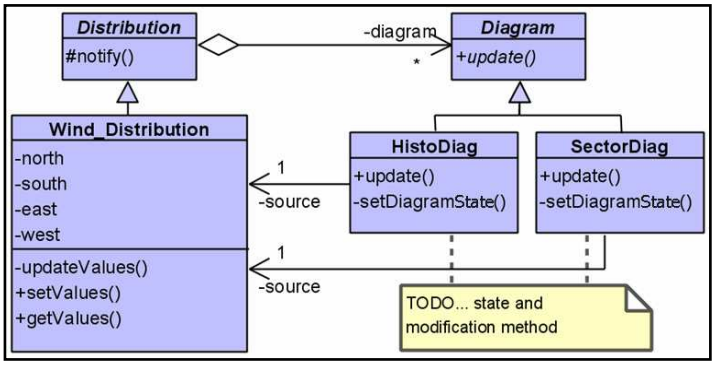


Figure 10. *Vue statique d'un modèle imité [adapté]*

5. Travaux liés

Les travaux visant à améliorer la spécification des solutions des patrons afin de garantir une réutilisation plus sûre peuvent être caractérisés de différentes manières. Nous retiendrons ici trois critères d'amélioration.

Complétude des spécifications. Dans la plupart des travaux (Meijler et al., 1997), (Arnaud et al., 2004), seuls les aspects statiques (classes, propriétés et associations) sont pris en compte. Dans une approche comme (Albin-Amiot et al., 2001), certains apports dynamiques sont mis en œuvre, grâce à un méta-modèle plus spécifique aux patrons que UML. Par exemple, la méta-classe *PDelegatingMethod* (spécialisation de la méta-classe *Method*) est utilisée pour spécifier qu'une méthode fait appel à une autre via une association donnée. Cela permet, entre autre, une certaine automatisation de la génération de code. Dans (France et al., 2004) les vues statiques (Structural Pattern Specification) et dynamiques (Interaction Pattern Specification) sont traitées conjointement, les IPS spécifiant les interactions des participants décrits dans les SPS. Notre proposition manipule et adapte des spécifications complètes, comportant trois types d'aspects : fonctionnels, dynamiques et statiques.

Expression de la variabilité. (Budinsky et al., 1996), (Sunyé, 1999) et (Le Guennec *et al.*, 2000) traitent de l'expression et du choix de variantes d'implémentation de patron. On peut positionner une bonne partie des apports de ces travaux, au niveau de ce que nous appelons la *réduction*. Si l'expression de la variabilité a peu été utilisée dans les travaux sur les patrons, rappelons qu'elle a par contre été utilisée dans d'autres contextes tels que les lignes de produit (Ziadi et al., 2005) ou l'ingénierie des besoins (Bennasri, 2005).

Extension du méta-modèle d'UML. Nous avons présenté une partie des extensions d'UML nécessaires à notre approche afin d'exprimer la variabilité et des règles de généricité. Nous définissons un méta-modèle commun à tous les patrons alors que dans beaucoup d'approches (Meijler et al., 1997), (France et al., 2004) le méta-modèle d'UML est étendu par des concepts spécifiques à chaque patron.

6. Conclusion

Nous proposons dans cet article un processus de réutilisation pour les patrons de conception, appelé processus d'imitation. Il tient compte à la fois des aspects variables mais également génériques de la solution proposée par l'ingénieur de patrons. Pour permettre l'expression de telles propriétés, nous utilisons la multi-modélisation en proposant à l'ingénieur de patrons de décrire, pour chaque patron, un mini-système comportant des vues fonctionnelle, dynamique et statique. Pour spécifier la variabilité et la généricité de ce mini-système, nous étendons UML afin d'articuler le modèle imitable (solution du patron) ainsi que le processus d'imitation autour des points de variation et des variantes (section 2).

Le processus de réduction (section 3) est la première partie du processus d'imitation. Il permet au concepteur de systèmes d'information de transformer un modèle imitable à variantes en modèle imité (dans l'état adaptable) à partir du choix des variantes. Le modèle imité [adaptable] est dépourvu de toute variabilité mais comporte encore les informations génériques nécessaires à la deuxième partie du processus : le processus d'application.

Le processus d'application (section 4) prend en compte le contexte de réutilisation pour adapter, résoudre et intégrer le modèle imité tout en respectant les contraintes de généricité exprimées par l'ingénieur de patrons.

Dans cet article, nous avons défini un cadre fonctionnel précis pour le processus d'imitation et devons maintenant compléter notre étude en insistant plus particulièrement sur l'expression de la généricité statique et l'instrumentation des transformations successives de modèles.

7. Bibliographie

Albin-Amiot H., Guéhéneuc Y.G., « Meta-modeling Design Patterns : application to pattern detection and code synthesis », *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*, June 2001.

Alexander C., *The Timeless Way of Building*, Oxford University Press, 1979.

Arnaud N., Front A., Rieu D., « Deux opérateurs pour l'intégration d'imitations de patrons », *Congrès INFORSID'05*, Mai 2005.

Arnaud N., Front A., Rieu D., « Une approche par méta-modélisation pour l'imitation des patrons », *Congrès INFORSID'04* , Mai 2004.

Bachmann F., Bass L., « Managing variability in software architecture », *ACM SIGSOFT Software Engineering Notes*, Volume 26, n°3, Mai 2001

Bennasri S., Une approche intentionnelle de représentation et de réalisation de la variabilité dans un système logiciel, Thèse de doctorat, Université de Paris I, Février 2005.

Budinsky, F.J., M.A. Finnie, J.M. Vlissides et P.S. Yu, « Automatic code generation from design patterns ». *IBM Systems Journal*, 1996

Czarnecki K., Eisenecker U. W., *Generative Programming – Methods, Tools and Applications*, Addison-Wesley, 2000.

France R.B., Dae-Kyoo K., Sudipto G., Eunjee S., « A UML-Based Pattern Specification Technique », *IEEE transactions on software engineering*, vol. 30, no. 3, March 2004

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns : Element of Reusable Object-Oriented Software*, Addison-Wesley professional computing series, 1995.

Le Guennec A., Sunyé G., Jézéquel J.M., « Precise modeling of design patterns », in *proceedings of UML 2000*, volume 1939 of LCNS, pages 482--496. Springer Verlag, 2000.

Object Management Group, « Unified Modeling Language : Superstructure », version 2.0, Août 2005.

Sunyé, G., « Génération de code à l'aide de patrons de conception », *Langages et Modèles à Objets - LMO'99*, Villefranche s/ mer, 1999.

Van der Maβen T., Lichter H., « Modeling variability by UML use case diagrams », *International Workshop on Requirements Engineering for Product Lines (REPL'02)*, pages 19-25. AVAYA labs, Septembre 2002.

Ziadi T, Jézéquel J.M., « Manipulation de lignes de produits logiciels : une approche dirigée par les modèles », *Ingénierie Dirigée par les Modèles (IDM'05)*, Mai 2005.

Processus de Modélisation Incrémentaux

Raphaël Marvie — Mirabelle Nebut

Laboratoire d'Informatique Fondamentale de Lille
UMR CNRS 8022
Université des Sciences et Techniques de Lille
Bâtiment M3 – UFR d'IEEA
F-59655 Villeneuve d'Ascq
{marvie,nebut}@lifl.fr

RÉSUMÉ. *Dans le cadre de l'ingénierie dirigée par les modèles (IDM), la constuction de logiciels repose sur la définition de modèles bien formés et suffisamment complets pour être exploitables. Cependant, modéliser n'est pas une activité simple. Pour que l'IDM profite pleinement au plus grand nombre, il est important de la rendre accessible à des non spécialistes. Motivés par la nécessité de structurer et d'encadrer l'activité de modélisation, nous proposons dans cet article un cadre de travail pour la définition et l'outillage de micro-processus de modélisation, appelés* processus de modélisation incrémentaux *(PMI). Un PMI décompose l'activité de modélisation en étapes : chaque étape permet à l'utilisateur de se concentrer sur un seul aspect de son modèle et l'enchaînement des étapes est défini par un automate. Des contrats associés aux étapes permettent de contrôler que le modèle est correct et suffisamment complet par rapport aux besoins.*

ABSTRACT. *Dealing with Model Driven Engineering (MDE), the building of software applications relies on the definition of models that need to be both well-formed and complete enough regarding their expected use. However, modelling is not a trivial activity. In order for MDE to benefit to a wide range of users, it is important to make it available to non specialists. Encouraged by the need for structuring and assisting the modelling, we propose in this paper a framework for the definition and tooling of modelling micro-processes, named* Incremental Modelling Processes *(IMP). Such a process defines the modelling as a set of steps: Each step permits the user to focus on a single aspect of his/her model at a time, and the flow of steps is defined as an automaton. Contracts associated with the steps permit the control of the model definition (correctness and completion regarding the objectives).*

MOTS-CLÉS : *micro-processus de modélisation, (méta-)modélisation incrémentale, assistance à la modélisation*

KEYWORDS: *modelling micro-process, incremental (meta)modelling, modelling support.*

1. Introduction

Dans le cadre de l'ingénierie dirigée par les modèles (IDM), la modélisation d'une application suit le plus souvent un *macro-processus* [KEN 02]. Des approches bien connues comme le processus unifié [JAC 99] ou SPEM [OMG 02] permettent de définir de tels macro-processus, imposant par exemple de commencer par définir des cas d'utilisation, suivis de scénarios et diagrammes de séquences, etc. La définition d'un modèle particulier lors d'une étape du macro-processus relève elle du *micro-processus* [KEN 02]. Il semble que les micro-processus aient été peu étudiés et peu outillés jusqu'à présent, alors que leur utilisation nous semble importante pour deux raisons : d'une part l'activité de modélisation n'est pas triviale quand on ne dispose que d'un méta-modèle comme tout mode d'emploi ; d'autre part il est tout à fait possible de concevoir un modèle conforme à son méta-modèle mais inutilisable pour l'exploitation pour laquelle il était initialement destiné. Nous détaillons maintenant ces deux aspects, motivations du travail présenté dans cet article.

Modéliser n'est pas une activité simple. Il suffit pour le constater d'ouvrir un modeleur standard pour créer un diagramme de classes : l'outil se contente de proposer en vrac toutes les fonctionnalités qui permettent de manipuler le modèle. L'utilisateur est noyé sous la masse des possibilités : ne sachant par où commencer, il risque d'être confronté au syndrome de la page blanche. Dans le meilleur des cas le méta-modèle est fourni avec des «guidelines» ou «best practices» écrits en langue naturelle. Ces textes servent de mode d'emploi à la modélisation : "commencez par identifier les composants, puis les points de connexion, et ensuite les services disponibles pour ces points, etc". Ils jouent le rôle de micro-processus informels, non outillés. Ce manque de structuration dans l'activité de modélisation est particulièrement préjudiciable quand l'utilisateur n'est pas un expert en modélisation, ce qui est de moins de moins rare au fur et à mesure que l'IDM passe dans les mœurs. C'est le cas par exemple quand un professeur d'anglais doit modéliser un scénario pédagogique afin de le déployer automatiquement sur une plate-forme d'enseignement à distance (EAD). Il nous semble que, pour que l'IDM profite pleinement au plus grand nombre, chaque communauté utilisatrice (par exemple celle de l'EAD) devrait pouvoir définir et outiller facilement ses propres micro-processus, de telle sorte que l'utilisateur lambda puisse ensuite être assisté et guidé dans sa construction d'un modèle.

Motivés par la nécessité de structurer et d'encadrer l'activité de modélisation, nous proposons dans cet article un cadre de travail pour la définition et l'outillage de micro-processus, appelés *processus de modélisation incrémentaux* (PMI). Un PMI permet à un utilisateur de construire incrémentalement un modèle (actuellement un diagramme structurel) tout en étant assisté dans sa démarche. Pour ce faire un processus est constitué d'un ensemble d'étapes, chaque étape de modélisation étant dédiée à une préoccupation particulière du modèle. Une étape ne met à la disposition de l'utilisateur qu'un nombre limité de concepts (et relations) du méta-modèle global : il lui est associé un méta-modèle partiel. D'autre part une étape n'offre qu'un sous-ensemble des opérations de modélisation permettant de construire le modèle. La dynamique du processus

est définie par un automate dont les transitions contraignent et contrôlent les enchaînements possibles entre étapes.

Par ailleurs, on ne crée pas des modèles dans l'absolu mais dans l'objectif d'une utilisation précise. Un même méta-modèle peut être utilisé de manière spécifique pour deux objectifs différents. Par exemple, un diagramme de classes décrivant une interface graphique n'est pas soumis aux mêmes contraintes qu'un diagramme de classes décrivant une base de données, pourtant ils sont conformes au même méta-modèle (celui d'UML). Pour être exploitable un modèle doit bien sûr être correct (par exemple ne pas contenir de relation d'héritage cyclique) mais aussi complet, dans le sens de suffisamment détaillé pour l'objectif visé (par exemple permettre une production de code automatique par transformation ou génération). Dans l'exemple de l'interface graphique on vérifiera qu'il existe au moins un formulaire de saisie et une zone d'affichage, dans celui de la base de données qu'il existe au moins une classe avec un attribut pour stocker les données. Les PMI que nous proposons permettent de contrôler au fil de la modélisation la correction et la complétude du modèle construit, vis à vis de critères spécifiés par le concepteur du PMI et énoncés par des *contrats* (au sens du *Design By Contract™* [MEY 92]).

Il est important de noter que l'utilisation d'un PMI n'est pertinente que dans le cas où un expert du domaine visé possède une démarche de conception éprouvée et souhaite en faire bénéficier, en la matérialisant par un outil d'assistance, des concepteurs non spécialistes de ce domaine. Évaluer la qualité de cette démarche n'est pas du ressort de notre problématique : les PMI visent uniquement à fournir des moyens de structurer et outiller une démarche donnée. Nous ne prétendons pas que les PMI proposés pour les méta-modèles illustrant cet article (qui ne sont par ailleurs pas des contributions en eux-mêmes) sont les seuls micro-processus possibles, ni que ce sont les meilleurs. Il s'agit uniquement de démarches de conception parmi d'autres.

Cet article est organisé comme suit. Les principes fondamentaux des processus de modélisation incrémentaux sont présentés en section 2 à travers deux exemples qui ont été choisis pour leur simplicité et la complémentarité des aspects qu'ils illustrent. La section 3 présente comment les méta-modèles associés aux étapes sont définis par raffinement. Enfin la section 4 explique comment sont définies les opérations de modélisation. Nous concluons cet article par une discussion de travaux connexes à notre proposition en section 5, puis par des perspectives de ce travail (Section 6).

2. Principes des processus de modélisation incrémentaux

L'objectif de l'activité de modélisation est de construire un modèle conforme à un méta-modèle pour un objectif spécifique. Les PMI visent à structurer cette activité de modélisation en étapes, de telle sorte qu'une étape corresponde à une sous-activité particulière de la modélisation : d'une part il lui est associé un méta-modèle spécifique qui est potentiellement un sous-ensemble du méta-modèle global (n'offrant pas tous les concepts et relations du domaine de modélisation), d'autre part elle n'offre

qu'un sous-ensemble des opérations proposées pour construire le modèle. Le méta-modèle des PMI est donné figure 1. Nous l'illustrons ici à partir de deux exemples de construction de modèle structurel : le premier pour une application avec le patron MVC (section 2.1) et le second pour une application de type EJB3 par annotation d'un diagramme de classes UML (section 2.2).

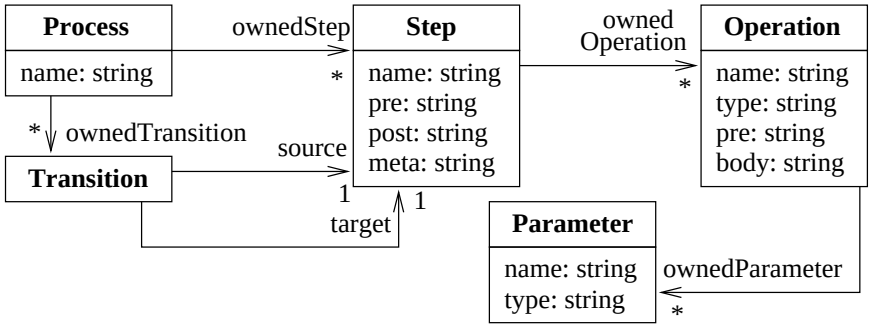


Figure 1. *Le méta-modèle des processus de modélisation incrémentaux*

2.1. *Exemple de la modélisation d'applications avec MVC*

Cet exemple suit le patron de conception Modèle / Vue / Contrôleur (MVC). La construction du modèle structurel, dont le méta-modèle associé (appelé `mmMVC`) est donné partiellement figure 2, représente une étape d'un macro-processus comprenant la définition de cas d'utilisation, la conception en suivant le patron MVC (en associant un contrôleur à chaque acteur du diagramme de cas d'utilisation et une opération à chaque action), la projection vers une technologie particulière et la mise en œuvre de l'application dans cette technologie.

2.1.1. *Structuration en étapes*

Même s'il n'est pas difficile à comprendre, le méta-modèle `mmMVC` est suffisamment conséquent pour qu'un besoin de structuration se fasse sentir : on ne sait pas trop par où attaquer la modélisation et plusieurs démarches sont possibles. Nous avons choisi de structurer l'activité de modélisation simplement en trois étapes (`Step`) séquentielles, à titre d'exemple. Le processus de modélisation est donc défini par un automate à 3 étapes (3 états plus un état initial et un état final) dont les transitions obligent à enchaîner les trois étapes en séquence, comme le montre la figure 3. Les notions de processus, états et transitions de l'automate correspondent respectivement aux concepts `Process`, `Step` et `Transition` dans le méta-modèle des PMI de la figure 1.

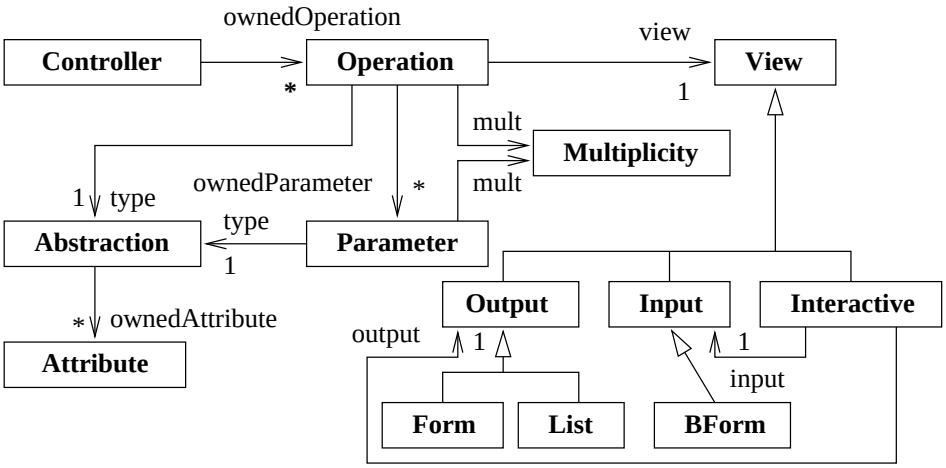


Figure 2. *Méta-modèle* `mmMVC` *de l'exemple MVC (extrait)*

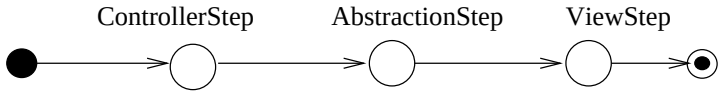


Figure 3. *Structuration en étapes pour l'exemple MVC*

2.1.2. *Description de la première étape*

Nous décrivons ici la première étape du processus, appelée `ControllerStep`, en mettant en valeur les aspects fondamentaux des PMI : structuration du méta-modèle et des opérations de modélisation associés au processus, ainsi qu'utilisation de contrats.

2.1.2.1. Structuration du méta-modèle global

L'étape `ControllerStep` est définie figure 4 dans une syntaxe propre à notre prototype. Cette première étape du processus permet d'entamer la construction d'un modèle conforme au méta-modèle `mmMVC`. Lors de cette étape nous proposons de définir les contrôleurs `Controller` et de leur associer des opérations `Operation` mais en ne précisant pour le moment que le nom des opérations (extraction des informations contenues dans les cas d'utilisation[1]). Les concepts du méta-modèle `mmMVC` ne sont donc pas tous nécessaires : il suffit d'associer à l'étape le méta-modèle `Step1mmMVC` décrit figure 4, sous-ensemble du méta-modèle `mmMVC` qui ne contient que les concepts `Controller` et `Operation`. Cette association d'un méta-modèle partiel à une étape correspond à l'attribut `meta` de `Step` dans le méta-modèle des PMI donné figure 1.

1. Cette étape peut être automatisée si le diagramme de cas d'utilisation est réifié.

```
step ControllerStep {
  # define the controller part of the model
  meta is Step1mmMVC ;
  pre: True ;
  post: len(model.Controller) > 0 and
        forall c in model.Controller, len(c.ownedOperation)  > 0 ;
  operation create_controller(String name) : Controller {
      pre: not exist c in model.Controller, c.name == name ;
  }
  operation add_operation(String name, Controller c) : Operation {
      pre: not exist o in c.ownedOperation, o.name == name ;
  }
}
metamodel Step1mmMVC {
  primitive String ;
  class Controller {
    attribute name : String ;
    attribute ownedOperation : Operation [0..*] ;
  }
  class Operation {
    attribute name : String ;
  }
}
```

Figure 4. *Définition de l'étape* `ControllerStep`*, exemple MVC (extrait)*

2.1.2.2. Structuration des opérations de modélisation offertes par le processus

En accord avec le méta-modèle `Step1mmMVC`, les opérations de modélisation offertes par l'étape `ControllerStep` sont la création de `Controller` et de `Operation` rattachées à un contrôleur. Une étape n'offre qu'un sous-ensemble de l'ensemble de toutes les opérations offertes par le processus (ce qui correspond aux éléments `Operation` et `ownedOperation` du méta-modèle des PMI).

2.1.2.3. Contrats associés au processus

Les opérations et les étapes d'un PMI possèdent des contrats (attribut `pre` de `Operation` et `pre` et `post` de `Step` dans le méta-modèle des PMI). Les opérations possèdent un corps (attribut `body`) qui sera détaillé en section 4. Les pré-conditions des opérations servent à éviter l'introduction d'incohérences dans le modèle, par exemple `create_controller` empêche de créer deux `Controller` de même nom. La pré-condition d'étape doit être vraie pour qu'il soit possible d'entrer dans cette étape, et la post-condition doit être vraie pour qu'il soit possible de sortir de l'étape en la terminant[2]. En tant qu'étape initiale `ControllerStep` ne possède pas de pré-condition. Sa

2. Une post-condition d'étape interprétée avec la vue classique de [MEY 92] indique que l'étape *garantit* sa post-condition quand le processus quitte l'état associé. Nous utilisons plus souvent

post-condition interdit de quitter l'étape tant qu'il n'existe pas au moins un contrôleur et que tout contrôleur ne possède pas au moins une opération. On assure ainsi une certaine complétude du modèle par rapport aux besoins exprimés. L'expressivité des contrats est celle de la logique du premier ordre.

2.1.3. *Description des étapes suivantes*

La seconde étape `AbstractionStep` du processus de modélisation permet de définir les abstractions manipulées par un contrôleur et de compléter la définition des opérations d'un contrôleur sur la base des abstractions créées. Nous ne donnons pas la description textuelle de cette étape par manque de place. Le méta-modèle `Step2mmMVC` associé à l'étape enrichit le méta-modèle `Step1mmMVC` (suivant un mécanisme de raffinement décrit en section 3) en lui ajoutant d'une part les concepts reliés aux abstractions `Abstraction` et `Attribute` et d'autre part les concepts `Parameter` et `Multiplicity` ; et en enrichissant par ailleurs la définition de `Operation` avec son type de retour `type` de multiplicité `mult` et ses paramètres `ownedParameter`. Les opérations offertes par l'étape permettent de créer des `Abstraction` (en empêchant des redondances de nommage) et des `Attribute` pour ces abstractions, d'ajouter des paramètres à une `Operation` et de fixer son type de retour. Il n'est plus possible de créer des `Controller` comme dans l'étape `ControllerStep`, bien que le concept `Controller` apparaisse dans le méta-modèle de l'étape `AbstractionStep`. Si l'utilisateur se rend compte qu'il lui manque un contrôleur à cette étape, il devra entamer une nouvelle itération[3].

La pré-condition de l'étape impose de n'y entrer que si un `Controller` a déjà été créé (dans le cas contraire il ne sert à rien dans le contexte de ce processus de créer des abstractions). Dans ce processus la post-condition de l'étape `ControllerStep` implique la pré-condition de l'étape `AbstractionStep`[4]. Si l'étape `ControllerStep` ne permettait pas de valider la pré-condition de l'étape `AbstractionStep` le processus serait absurde et inutilisable. Assurer que le processus est bien conçu et ne contient pas d'inter-blocage est actuellement à la charge du concepteur du PMI (cf Section 6).

Quand l'utilisateur demande à quitter l'étape `ControllerStep`, deux cas de figure peuvent se produire : soit la post-condition n'est pas vérifiée et le processus reste dans l'état courant pour donner la possibilité de faire évoluer le modèle ; soit la post-condition est vérifiée et l'étape est validée, puis le processus entre dans l'état

---

l'interprétation duale : il est *interdit* de quitter l'étape tant que la post-condition n'est pas vérifiée (il faut encore appliquer au moins une opération pour modifier l'état du modèle).

3. La notion d'itération (au sens habituel du génie logiciel) n'est pas traitée dans ce papier. Elle correspond au déroulement d'un PMI aboutissant à un modèle partiel (par exemple, lors de la première itération seules quelques fonctionnalités sont modélisées). Lors d'une itération suivante (par exemple pour l'ajout de nouvelles fonctionnalités) les éléments de modèles produits par les itérations précédentes ne sont pas perdus et sont manipulables au même titre que les éléments de modèles créés dans l'itération courante. Un modèle n'est considéré conforme au PMI que s'il est issu de son étape finale, donc que la dernière itération a été complète.

4. Plus généralement la pré-condition d'une étape doit être impliquée par une composition des post-conditions des étapes précédentes.

`AbstractionStep`. L'utilisateur a toujours le choix d'annuler une étape non encore validée en retournant dans l'état précédent. L'atomicité des étapes assure que le modèle ne sera pas laissé dans un état incohérent : soit une étape est totalement validée, soit aucune des opérations effectuées dans l'étape n'est prise en compte.

La troisième étape (qui permet la définition des vues associées aux contrôleurs) n'est pas détaillée ici.

2.2. *Exemple de la modélisation d'applications de type EJB3*

Cet exemple concerne la conception d'applications de type EJB3 par annotation d'un diagramme de classes UML. La construction d'un modèle structurel représente une étape d'un macro-processus comportant la définition du diagramme de classes et la définition de composants EJB3, puis la projection vers le langage Java 1.5 et la mise en œuvre de l'application.

Le processus que nous proposons comporte quatre étapes et n'est pas linéaire (cf figure 5). La première étape `ClassStep` permet la réalisation d'un diagramme de classes UML. Le processus ne propose pas de structurer cette étape (qui mériterait un processus en elle-même), le but étant de se concentrer sur la définition des composants EJB (on peut aussi charger un diagramme de classes existant). Le méta-modèle associé à l'étape est celui d'UML pour les diagrammes de classe avec stéréotype (classes, associations, attributs, opérations, stéréotypes).

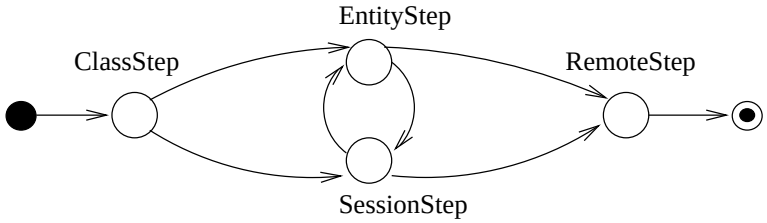


Figure 5. *Structuration en étapes pour l'exemple EJB3*

Les deux étapes «suivantes» `EntityStep` et `SessionStep` permettent d'identifier les composants de type entité (persistance des données) et session (traitement sur ces données) respectivement, notamment par l'association aux classes concernées d'un stéréotype `entity` et `session` respectivement. La dernière étape `RemoteStep` permet l'identification des composants accessibles à distance (qu'ils soient de type entité ou session), notamment par l'association aux classes concernées d'un stéréotype `remote`.

Chaque étape utilise les mêmes concepts que ceux utilisés par la première étape `ClassStep` : toutes les étapes partagent le même méta-modèle, même si chaque étape est dédiée à une préoccupation particulière. Un processus de modélisation ne structure donc pas nécessairement le méta-modèle global. Par contre, comme pour l'exemple

MVC de la section 2.1, chaque étape n'offre qu'un sous-ensemble de toutes les opérations offertes par le processus : les étapes qui suivent `ClassStep` ne permettent que l'ajout de stéréotypes.

La dynamique du processus est plus complexe que celle de l'exemple MVC. Si les étapes `ClassStep` et `RemoteStep` viennent bien respectivement avant et après toutes les autres, les deux étapes `EntityStep` et `SessionStep` sont indépendantes et les opérations qu'elles offrent peuvent être enchaînées dans n'importe quel ordre. La post-condition de `ClassStep` (existence d'au moins une classe possédant soit un attribut soit une opération) implique la disjonction des pré-conditions de `EntityStep` (existence d'au moins une classe possédant un attribut) et `SessionStep` (existence d'au moins une classe possédant une opération). Quitter `ClassStep` permet donc d'entrer dans `EntityStep` comme dans `SessionStep`. Il est ensuite possible d'aller et venir entre les étapes `EntityStep` et `SessionStep`.

3. Définition de méta-modèles par raffinement

Cette section explique comment les méta-modèles associés aux étapes d'un processus sont définis par un mécanisme de *raffinement*. Dans ce travail, et conformément à [MAR 05], raffiner un méta-modèle consiste à lui ajouter de nouveaux éléments ou à enrichir la définition d'éléments existants.

3.1. *Support de méta-modélisation*

Les méta-modèles manipulés dans le cadre de ce travail sont conformes à E-MOF (*Essential MOF*) qui représente le cœur de la norme MOF 2.0 [OMG 04]. Nous avons choisi de nous conformer à E-MOF car ce sous-ensemble du *Meta Object Facility* offre un jeu de concepts réduit mais suffisant pour définir des méta-modèles. Rappelons que chaque méta-modèle E-MOF est défini par un paquetage. Nos expérimentations sont basées sur l'outil PyEMOF qui implémente E-MOF en Python. Cet outil permet la génération de dépositaires de modèles (*model repositories*), ainsi que l'import/export de fichiers XMI associés, à partir de la définition d'un méta-modèle conforme à E-MOF. Il implémente aussi certaines opérations comme le `merge` ou la relation d'importation de méta-modèles. Pour faciliter l'écriture de méta-modèles, nous utilisons une syntaxe textuelle propre à PyEMOF (mais très proche de syntaxes existantes comme celle de KerMeta [MUL 05]).

3.2. *Principes du raffinement*

Le raffinement de méta-modèle repose sur l'utilisation de deux relations que nous avons nommées `refines` et `using`. Nous expliquons les principes de ces deux relations en les présentant sur l'exemple de la section 2.1 (MVC), comme illustré par la figure 6 dont la syntaxe textuelle est donnée figure 7. Un nouveau méta-modèle (par

exemple `Step2mmMVC`) est défini sur la base d'un ou plusieurs méta-modèles existants (dans ce cas `Step1mmMVC`) en utilisant (éventuellement) des concepts provenant d'un ou plusieurs méta-modèles tiers (dans ce cas `abstraction`).

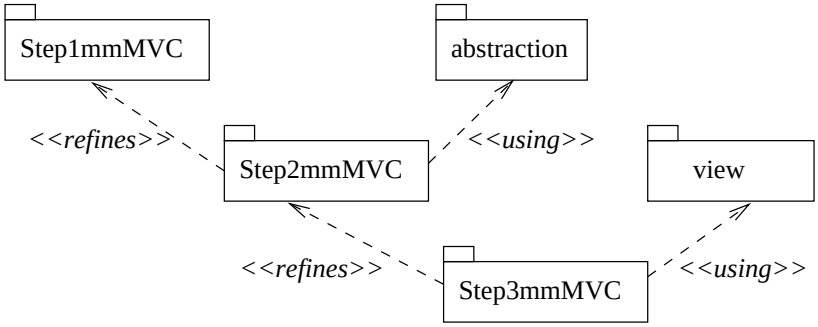


Figure 6. *Définition d'un méta-modèle par raffinement*

```
metamodel abstraction {
  class Abstraction { # definition des attributs }
  # definition d'autres classes
}

metamodel Step2mmMVC refines Step1mmMVC using abstraction {
  class Multiplicity { # ... }
  class Parameter {
    attribute name : String ;
    attribute type : Abstraction ;
    attribute mult : Multiplicity ;
  }
  class Operation {
    attribute type : Abstraction ;
    attribute mult : Multiplicity ;
    attribute ownedParameter : Parameter ;
  }
}
```

Figure 7. *Exemple de raffinement de méta-modèle (extrait)*

La relation `refines` entre deux méta-modèles représente l'axe principal du processus de modélisation. Le méta-modèle `Step2mmMVC` de la figure 6 représente une évolution du méta-modèle `Step1mmMVC` par ajout de nouveaux éléments (par exemple `Parameter`) ou par enrichissement de la définition des éléments existants (par exemple ajout des paramètres et du type de retour comme attributs d'`Operation`). La relation de raffinement est mise en œuvre par la relation `merge` du MOF. Par défini-

tion, tous les éléments définis dans le méta-modèle `Step1mmMVC` se retrouvent donc dans `Step2mmMVC` (comme s'ils étaient définis dans ce dernier). De plus, quand deux classes définies dans `Step1mmMVC` et `Step2mmMVC` portent le même nom (par exemple `Operation`) alors elles sont considérées comme une seule classe et sont fusionnées (les attributs de la version de base sont ajoutés aux attributs de la version raffinée).

La relation `using` représente l'intégration de préoccupations au sein d'un méta-modèle. L'utilisation par `using` de `abstraction` dans le raffinement de `Step1mmMVC` indique que les classes définies dans le méta-modèle de base pourront être enrichies avec des concepts (classes) provenant du méta-modèle `abstraction`. Par contre il n'est pas possible d'enrichir la définition d'éléments de `abstraction` dans `Step2mmMVC`. Ces définitions seront toutefois utilisables dans un raffinement ultérieur comme des définitions de base. La relation d'intégration de préoccupations est mise en œuvre par la relation `import` du MOF.

Nous avons choisi d'utiliser les deux relations `refines` et `using` plutôt que la seule relation `merge` (de bas niveau) pour faire la distinction entre l'axe de définition du processus de modélisation et les préoccupations qui sont intégrées progressivement au sein des modèles. Dans les cycles de développement en Y, nous avons dans la majorité des cas une approche non symétrique : un des axes du Y est «principal» et l'autre est «secondaire». Dans le cadre de nos expérimentations, le point de départ (le méta-modèle initial) représente en général le domaine d'application sur lequel nous tissons progressivement différentes préoccupations.

4. Des opérations pour outiller les étapes

Nous présentons dans cette section comment sont définies les opérations de modélisation associées aux étapes d'un processus. Une opération comporte une pré-condition (`pre`) et un comportement (`body`).

Dans le prototype actuel, la pré-condition et le comportement sont définis dans une syntaxe propre qui est ensuite projetée en Python. Dans un contexte tout UML le comportement peut être défini en Action Semantics [OMG 01] et les pré-conditions en OCL. Dans un contexte MOF, le comportement peut être défini en Kermeta, ce sur quoi porteront nos prochaines expérimentations.

L'ensemble des opérations associées à une étape d'un PMI est défini par son concepteur et spécifie strictement comment l'utilisateur peut manipuler le modèle pendant cette étape. Il ne représente qu'un sous-ensemble de toutes les manipulations applicables au modèle pendant l'ensemble du processus. Nous considérons deux sortes d'opérations : les opérations *basiques* et les opérations *composites*.

Les opérations basiques peuvent être vues comme l'«assembleur» de la manipulation de modèles. Elles correspondent aux opérations CRUD (création, destruction, mise en relation et suppression d'une relation) augmentées du positionnement d'attribut de type primitif. L'ensemble des opérations basiques associé à une étape est

généré automatiquement sur la base du méta-modèle de cette étape : à chaque concept sont associées les deux opérations de création / destruction, à chaque attribut de type construit ou association entre deux concepts sont associées deux opérations pour la mise en relation et la suppression de cette relation, à chaque attribut de type primitif est associée une opération de positionnement. Les opérations basiques ont un grain trop fin pour être utilisables en pratique. La simple création d'un contrôleur (cf `create_controller` en section 2.1.1) fait intervenir les deux opérations basiques que sont la création du contrôleur et le positionnement de son attribut `name`. Ces opérations ne sont donc jamais offertes directement à l'utilisateur, qui utilisera des opérations composites.

Une opération composite est définie comme une composition d'opérations. Son comportement est défini par une spécification exécutable écrite dans un petit langage inspiré de Action Semantics et de KerMeta. Les instructions élémentaires sont les opérations basiques et l'invocation d'opération composite. Les structures de contrôle sont la séquence, l'itération, la conditionnelle et le `return`. À titre d'exemple le corps des opérations de la figure 4 est donné figure 8. Certaines opérations retournent une valeur : il est en effet souvent nécessaire d'avoir accès aux objets créés par une opération pour la tester, ou quand on souhaite utiliser des opérations composites pour définir des opérations de plus haut niveau.

```
operation create_controller(String name) : Controller {
  pre: not exist c in model.Controller, c.name == name ;
  body: c = create Controller ;
        c.name = name ;
        return c
}
operation add_operation(String name, Controller c) : Operation {
  pre: not exist o in c.ownedOperation, o.name == name ;
  body: o = create Operation ;
        o.name = name ;
        link c.ownedOperation to o ;
        return o
}
```

Figure 8. *Définition des opérations de l'étape* `ControllerStep`*, exemple MVC*

La pré-condition d'une opération de modélisation est systématiquement vérifiée à l'appel de l'opération afin de contrôler sa bonne utilisation et de garantir la construction d'un modèle cohérent. L'expressivité des pré-conditions n'est pas limitée à la non-redondance des noms (exemples de la figure 8). La pré-condition de l'opération `create_multiple_output` donnée figure 9 indique qu'une vue destinée à afficher un ensemble d'éléments sous la forme d'une liste ne peut être associée à une opération qui ne retourne pas une séquence d'éléments, ni à une opération qui possède des paramètres. La pré-condition de l'opération `create_simple_interaction` indique que pour définir une vue d'interaction (formulaire de saisie et affichage) associée à

une opération d'un contrôleur, cette opération doit posséder au moins un paramètre (à saisir) et retourner une valeur (à afficher).

```
operation create_multiple_output(String name, Operation o) : View {
  pre: o.type != None and o.mult.upper == '*' and
       len(o.ownedParameter) == 0 ;
  body: v = create List ;
        v.name = name ;
        o.view = v ;
        return v
}
operation create_simple_interaction(String name, Operation o) : View {
  pre: o.type != None and o.mult.upper != '*' and
       len(o.ownedParameter) > 0 ;
  body: v = create Interaction ; v.name = name ;
        v.input = create BForm ; v.output = create Form ;
        o.view = v ;
        return v
}
```

Figure 9. *Définition d'opérations de l'étape* `ViewStep` *(extrait), exemple MVC*

5. Travaux connexes

Il n'existe pas à notre connaissance de travaux qui concernent les micro-processus de modélisation à proprement parler, même si le terme apparaît dans [KEN 02]. La construction incrémentale d'un modèle s'apparente à une suite de transformations de modèles, mais les travaux du domaine concernent les transformations automatiques, alors que dans notre cas elles sont manuelles. On notera que [CZA 03] ne mentionne pas dans sa classification la distinction entre transformations manuelles et transformations automatiques. La classification de [MEN 05], plus récente, fait cette distinction, mais en référençant des transformations type description des besoins en langue naturelle vers modèle d'analyse, ce qui ne correspond pas à notre problématique.

Le concept de raffinement de modèle a déjà été formalisé dans la littérature, mais son étude semble limitée aux modèles comportementaux, principalement les machines à états (cf par exemple [VAN 06] ou [LAM 04]). Dans ce cas le raffinement peut se définir formellement : un modèle C est un raffinement d'un modèle A si C préserve les comportements de A, c'est à dire que tout comportement de A est encore un comportement de C. Par ailleurs [EST 05] distingue informellement quatre catégories d'extension de méta-modèle : l'extension de comportement (modification du comportement existant dans un méta-modèle), l'extension de méta-modèle (ajout de nouveaux concepts), le raffinement (enrichissement de la définition d'un concept existant) et la composition (définition d'un nouveau méta-modèle en combinant deux méta-modèles existants). Notre définition du raffinement de méta-modèle, issue des travaux

de [MAR 04], regroupe les trois dernières catégories. L'approche basée sur l'intégration de préoccupations que nous utilisons peut être comparée à certains travaux sur la conception orientée aspects [AOS 05].

De nombreux travaux[5] sont menés relativement à la correction des modèles, vis à vis de critères reconnus qui peuvent s'appliquer à tous les modèles indépendamment du domaine de modélisation, comme l'absence de cycles dans les relations d'héritage. Les résultats concernent à la fois les propriétés de consistance intra-modèle (relatives à un unique modèle donné) et inter-modèle (relatives à différents modèles ou vues). Les résultats relatifs à la consistance intra-modèle pourraient être intégrés dans les PMI. L'originalité de notre approche tient au fait que c'est le concepteur du processus qui choisit ses propres critères de correction en les énonçant par des contrats. L'autre point original est que les contrats peuvent servir à énoncer des propriétés de complétude vis à vis de la future utilisation du modèle construit. Cette approche très souple ne peut être mise en œuvre par des critères globaux qui s'appliquent à tous les modèles.

6. Conclusion et perspectives

Cet article présente les processus de modélisation incrémentaux (PMI) que nous proposons dans le but de définir et outiller des micro-processus de modélisation. Notre principale contribution est de permettre la structuration des processus en étapes, en formalisant les enchaînements d'étapes possibles par un automate. À chaque étape est associé un méta-modèle qui n'est potentiellement qu'un sous-ensemble du méta-modèle global de l'application (jusqu'à présent via le mécanisme de raffinement de méta-modèle) : de ce fait le nombre de concepts manipulables et l'ensemble des opérations offertes lors d'une étape sont limités, réduisant ainsi la complexité ponctuelle de la modélisation. L'utilisation de contrats associés aux étapes et aux opérations permet de contrôler au fil de la modélisation que le modèle est correct et complet au regard de critères comme l'absence de deux éléments de modèles possédant le même nom ou la présence obligatoire de certains éléments de modèles.

À ce jour, nos expérimentations sur les PMI ont porté sur des méta-modèles et des modèles plus conséquents que les exemples jouets de cet article : méta-modèle proposé par l'organisme IMS-LD pour la modélisation de scénarios pédagogiques dans le cadre de l'EAD [PAL 06], méta-modèle d'applications interactives basées sur une variante du MVC utilisée en IHM. Dans ce dernier cas les modèles garantis complets par le PMI ont permis la génération automatique de l'ensemble du code non fonctionnel de l'application. On constate que plus le (méta-)modèle est gros, plus les avantages de la réduction de la complexité de la modélisation se font sentir à l'usage. De plus, l'assistance à l'usager est particulièrement appréciable pour des non spécialistes de la modélisation, voire pour des étudiants qui découvrent l'IDM.

---

5. Par exemple ceux présentés dans les ateliers de travail *Consistency Problems in UML-based Software Development* satellites de l'*International Conference on UML Modeling Languages and Applications «UML»*.

Les perspectives de ce travail sont de plusieurs ordres. Mêmes si les grandes lignes des PMI sont posées, certaines améliorations sont d'ores et déjà nécessaires. D'abord, un inconvénient du raffinement de méta-modèles est que le méta-modèle «raffinant» est par définition plus gros que le méta-modèle raffiné (il l'inclut sur le principe des poupées russes), ce qui n'est pas la solution la plus adaptée pour tous les processus. Nous pensons intéressant d'expérimenter en complément du raffinement un mécanisme de vues [CAR 03] du méta-modèle global de l'application. Pour les processus incluant des ruptures conceptuelles, par exemple le passage d'un modèle de cas d'utilisation à un modèle de type MVC, il est nécessaire de définir le macro-processus associé. Ce macro-processus est une séquence de micro-processus, tels que nous les avons présentés ici, et de transformations de modèles automatiques.

Ensuite, les automates représentant les PMI que nous avons expérimentés correspondent basiquement à des séquencements d'étapes et à des «mises en parallèle» d'étapes indépendantes (par exemple les étapes `EntityStep` et `SessionStep` de l'exemple EJB, Section 2.2). Le nombre de transitions entre étapes indépendantes croît exponentiellement avec le nombre d'étapes, rendant l'automate peu lisible. L'utilisation des Protocol State Machines d'UML 2.0 et de leurs différents types d'états peut sans doute améliorer ce point. Enfin, nous avons identifié que certains «types» d'opérations de modélisation sont récurrents. Nous pensons qu'il est possible de définir des opérations génériques de modélisation, d'un niveau plus abstrait que les CRUD. Nous comptons expérimenter ce point avec KerMeta et évaluer si le typage de modèle [STE 05] est une solution pour spécialiser des opérations génériques.

Le second volet de nos perspectives est plus formel. Les modèles construits en suivant un PMI sont validés par les contrats associés au processus, mais le processus lui-même doit aussi être validé. Un processus peut présenter de multiples sources d'incohérences, par exemple une configuration incluant une étape source qu'il est impossible de quitter même si elle possède des transitions sortantes, ou encore une étape cible non atteignable même si elle possède des transitions entrantes. Une perspective de ce travail est la détection de ces incohérences. L'idée serait d'associer à un PMI un système de transitions (formalisme couramment utilisé pour modéliser certains types d'application comme les systèmes réactifs) étiquetées par une combinaison des pré et post-conditions associées aux étapes. Pour spécifier les prédicats qui étiquettent les transitions, nous pensons utiliser une logique appartenant au formalisme des logiques de description (*DLs* [BAA 02]). Les DLs forment un sous-ensemble décidable de la logique du premier ordre et sont utilisées pour représenter des connaissances (ontologies, diagrammes de classes, etc) et leur appliquer des raisonnements.

7. Bibliographie

[AOS 05] AOSD.NET, « AOSD Web Page », 2005, `http://aosd.net/`.

[BAA 02] BAADER F., CALVANESE D., MCGUINNESS D., NARDI D., PATEL-SCHNEIDER. P., Eds., *The Description Logic Handbook : Theory, Implementation and Applications*, Cambridge University Press, 2002.

[CAR 03] CARON O., CARRÉ B., MULLER A., VANWORMHOUDT G., « A Framework for Supporting Views in Component Oriented Information Systems », *Proceedings of the International Conference on Object-Oriented Information Systems*, septembre 2003.

[CZA 03] CZARNECKI K., HELSEN S., « Classification of Model Transformation Approaches », *OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, 2003.

[EST 05] ESTUBLIER J., SANLAVILLE S., « Extensible Process Support Environments for Web Service Orchestration », *Proc. of the Intl. IEEE Conference on Next Generation Web Services Practices*, Seoul, Korea, August 2005.

[JAC 99] JACOBSON I., BOOCH G., RUMBAUGH J., *The Unified Software Development Process*, Addison-Westley Professional, 1999.

[KEN 02] KENT S., « Model Driven Engineering », BUTLER M., PETRE L., SERE K., Eds., *Proc. of the 3rd Intl. Conf. on Integrated Formal Method—IFM 2002*, vol. 2335 de *LNCS*, Turku, Finland, 2002, Springer-Verlag, p. 286-298.

[LAM 04] LAMBOLAIS T., GOUT O., « Using conformance relations to help the development of state-machines », *ISSRE'04, 15th IEEE International Symposium on Software Reliability Engineering*, november 2004.

[MAR 04] MARVIE R., « Vers des patrons de métamodèlisation, Structuration de métamodèles par séparation des préoccupations », *Technique et Science Informatique (TSI)*, vol. 23, n° 10, 2004, p. 1355-1382.

[MAR 05] MARVIE R., NEBUT M., « Un cadre de travail pour l'évolution contrôlée des modèles du logiciel », *Premier atelier de travail sur l'évolution du logiciel*, Berne, Suisse, Mars 2005.

[MEN 05] MENS T., VAN GORP P., « A taxonomy of model transformation », *Proc. Int'l Workshop on Graph and Model Transformation*, 2005.

[MEY 92] MEYER B., « Applying "Design by Contract" », *Computer*, vol. 25, n° 10, 1992, p. 40–51, IEEE Computer Society Press.

[MUL 05] MULLER P.-A., FLEUREY F., VOJTISEK D., DREY Z., POLLET D., FONDEMENT F., STUDER P., JÉZÉQUEL J.-M., « On Executable Meta-Languages applied to Model Transformations », *Model Transformations in Practice Workshop*, 2005.

[OMG 01] OMG, « Action Semantics for the UML », OMG Document ad/2001-08-04, 2001.

[OMG 02] OMG, « Software Process Metamodel Specification », rapport n° OMG TC Document formal/2002-11-14, 2002, Object Management Group.

[OMG 04] OMG, « Meta Object Facility (MOF) 2.0 Core Specification », OMG Document ptc/04-10-15, 2004.

[PAL 06] PALLEC X. L., MOURA O., MARVIE R., NEBUT M., TARBY J.-C., « Supporting Generic Methodologies to Assist IMS-LD Modelling », Kerkrade, Netherlands, juillet 2006.

[STE 05] STEEL J., JÉZÉQUEL J.-M., « Model typing for improving reuse in model-driven engineering », KENT S., BRIAND L., Eds., *Proceedings of MODELS/UML'2005*, vol. 3713 de *LNCS*, Montego Bay, Jamaica, octobre 2005, Springer.

[VAN 06] VAN DER STRAETEN R., JONCKERS V., MENS T., « A formal approach to model refactoring and model refinement », *Software Systems Modeling*, , 2006, Springer.

Gaining language independency in aspect-oriented design through meta-modeling and model transformation

Ouafa Hachani

Equipe SIGMA, LSR-IMAG
681, rue de la Passerelle, B.P. 72
38402 Saint Martin d'Hères cedex
Ouafa.Hachani@imag.fr

ABSTRACT. *Due to a certain lack of consensus on what are the basic aspect-oriented concepts and mechanisms, most aspect-oriented design modeling languages proposed today are specific to a particular aspect-oriented programming language (mainly AspectJ and Hyper/J). A more abstract modeling language is needed to fully express software designs in a language independent manner. We thus propose an extension of the UML meta-model for aspect-oriented concepts and mechanisms that allows for the expression of language-independent aspect-oriented designs. This meta-model can in its turn be furthermore extended to be language specific: we defined two of these extensions AspectJ/UML and HyperJ/UML that we also present in this paper. Transformation rules can then be applied for migrating models from a meta-model to another. Although our main motivation in this work was to provide an approach dedicated to the expression of design pattern structures (for which the language independency is an important issue), this approach can also be adopted in the case of general-purpose software designs.*

KEYWORDS: *aspect-oriented modeling, meta-modeling and model transformation, UML, AspectJ, Hyper/J, design patterns.*

1. Introduction

As it seems reasonable to admit in the first stages of a software development process that some general kind of programming paradigm will be used during the implementation phase, one should not have to definitely choose the programming language(s) too early in the design phase. Object-oriented design patterns as they are presented in (Gamma *et al.*, 1995), for instance, fulfil this idea: sample codes in several object-oriented programming languages (Smalltalk and C++) that can easily be adapted to even more programming languages are given for the same pattern, when a common structure of the pattern solution is expressed in terms of a modeling language with respect to consensual basic object-oriented concepts. Aspect-oriented programming languages still lack a consensus on their basic concepts that would be comparable in precision to the definitions of (Wegner, 1987) for example. On the one hand, the relative fuzziness in the commonly accepted definitions of what is a concern or an aspect, or what are aspect-oriented mechanisms encourages creativity in aspect-oriented programming language design. On the other hand, the actual diversity of today proposed aspect-oriented programming models and languages foster the broad acceptance of aspect-orientation. To gain more abstraction and become able to express aspect-oriented language independent structures of software designs, we adopt a meta-modeling approach. We chose to proceed by first providing language-specific meta-models. We then put them side by side for finding out similarities and consequently identify common abstract concepts, which we can use as the basis for the definition of a general meta-model that we propose for aspect-oriented modeling. We chose furthermore to adopt a model-to-model transformation approach in order to translate a language independent model that is an instance of the general meta-model to a corresponding model that is an instance of one language-specific meta-model. In this paper we propose two meta-models respectively specific to two of the most known aspect-oriented programming languages, AspectJ and Hyper/J. Both these meta-models and the more general one are defined as extensions of the UML 1.5 meta-model (OMG-UML, 2004). Transformation rules from a language-independent model to the instance of the AspectJ-specific or the Hyper/J-specific meta-model are also proposed.

Our motivation for the work we present in this paper come from a background researches that are doing on design patterns and aspect orientation. These works (Noda *et al.*, 2001), (Nordberg, 2001), (Hanneman *et al.*, 2002), (Hirschfeld *et al.*, 2003), as well as our own work (Hachani *et al.*, 2003 (a) et (b)) on this topic, propose to use aspect-oriented mechanisms to provide new implementations for object-oriented design patterns, improving their traceability and reusability. Nevertheless, because of a certain lack of consensus on the basic aspect-oriented concepts, the newly proposed solutions are only defined as language-specific implementations. They remain dependent of the use of the programming language for which they were proposed (most often AspectJ (Kiczales *et al.*, 2001), Hyper/J (Ossher *et al.*, 2000) and sometimes AspectS (Hirschfeld, 2002)), and their adaptation to another aspect-oriented language might not be straightforward. We

argue that more abstraction is needed to express design pattern aspect-oriented structures in a language independent manner, and this expression should include recommendations to project the abstract design into a specific aspect-oriented programming language. We thus need a modeling language to express aspect-oriented design. This language needs to encompass concepts that are abstract enough to express language independent structures, but it also need to be rich enough to ease the projection of those structures into language-specific designs.

The rest of this paper is organized as follows. Section 2 briefly recalls and discusses the extension mechanisms of UML. Section 3 introduces our general aspect-oriented meta-model by first presenting and then comparing both UML meta-model extensions that we propose for AspectJ and Hyper/J. Section 4 is devoted to our model transformation approach which is illustrated by the *Strategy* pattern. We consider some related works in section 5 and conclude this paper in Section 6.

2. Meta-modeling aspects and UML

Most of the currently proposed aspect-oriented programming languages build up on object-oriented programming languages. It is then quite natural to investigate UML suitability as a starting point to obtain a modeling language supporting aspect-orientation. UML (OMG-UML, 2004) is the OMG standard language for object-oriented design specifications. It is a general purpose modeling language that can be used in a wide range of application domains. As a compromise between the requirement for a standard and for domain-specific modeling, the UML was designed as an extensible modeling language. As such, it includes a set of built-in extension mechanisms (*stereotype*, *tagged value* and *constraint*) to extend or customize UML model elements. Another way for extending UML is the meta-modeling approach, which consist on extending the UML meta-model itself.

UML inherent extension mechanisms allow extending the UML by adding new building blocks, creating new properties, and specifying new semantics in order to make the language suitable for a specific domain. In this sense, stereotyping allows creating new model elements derived from existing ones, which are similar to the basic model elements, but that have specific properties and semantics. The problem with this first approach is that extension mechanisms might not meet every modeling need. Therefore, due to theirs restrictions, numerous domain-specific meta-models have been defined by extending the UML own meta-model with new meta-classes; this complementary approach is referred to as meta-modeling. In fact, extending strict object-orientation to aspect-orientation implies the definition of new concepts that are not directly related to object-oriented concepts. We thus consider the exclusive use of the extension mechanisms to be insufficient because it consists in more superficially approaching existing model elements instead of creating new ones with additional syntax, semantics and pragmatics. We thus chose to adopt a meta-modeling approach to work towards a general aspect-oriented extension of UML.

3. A general meta-model for aspect-oriented modeling

We propose in this section two extensions of the UML meta-model that are respectively specific to AspectJ and Hyper/J. We then propose a general meta-model that we elaborate on the basis of similarities found by comparing these two language-specific meta-models. The new model elements we propose and describe in the following subsections are an important addition to the UML meta-model. Therefore, we use the UML specification formalism (OMG-UML, 2004) to describe the syntax and semantics of these elements. The OMG defines UML using a meta-model which is described in a semi-formal manner, using three views: *abstract syntax*, *well-formedness rules* and *semantics*. Abstract syntax presents in a UML class diagram the meta-classes defining the concepts and their relationships. Well-formedness rules define the static semantics of the basic concepts by specifying constraints over their attributes and associations. Semantics defines the meaning of the constructs using natural language. In this paper, we mainly focus in the abstract syntax and partially in some well-formedness rules (see (Hachani, 2003) for more details).

3.1. *Extending the UML meta-model for AspectJ: AspectJ/UML*

In this subsection, we present the abstract syntax of the UML meta-model extension specific to AspectJ. Since AspectJ is built as an extension of the Java object-oriented programming language, AspectJ/UML includes all object-oriented fundamental concepts and relationships: Class, Interface, Attribute, Operation, Association, Generalization… In addition to these model elements, it also includes all AspectJ basic concepts and relationships (AspectJ, 2002): Aspect, Pointcut, Advice, Crosscutting… We detail some part of the definition of the corresponding model elements below (for a complete definition, see (Hachani, 2003)).

3.1.1. *Aspects*

Aspects provide convenient container for crosscutting features (Introduction, DeclareParents and Advice) and other crosscutting specifications elements such as Pointcuts. An aspect can also hold, like a class, Attributes and Methods. An aspect may implement zero or more interfaces, and extends zero or more aspects and/or classes; some aspects may be declared abstract. With respect to all this, aspects are special cases of classifiers (Figure 1). It is important to note for example that, an instance of the Aspect meta-class that contains at least one abstract pointcut has to be abstract. That kind of restrictions can't be easily expressed on a class diagram, but may nevertheless be specified by well-formedness rules using UML-OCL (*Object Constraint Language*) (OMG-OCL, 2003). We thus complete the abstract syntax by defining such rules for instances of the Aspect meta-class. Here is an example:

```
context Aspect inv: self.crosscuttingSpecification->exists ( cS : CrosscuttingSpecification|
                    cS.oclIsKindOf(NamedPointcut) and cS.isAbstract = #true)
                    implies self.isAbstract = #true
```

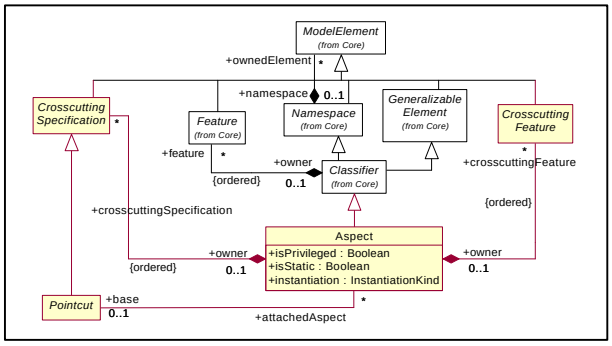


Figure 1. *Abstract Syntax of Aspect*

3.1.2. *Crosscutting features*

Crosscutting features (Figure 2) are owned by an aspect and are intended to affect some of the classifiers it crosscuts.

– Introduction. An introduction allows adding a new feature (attribute, operation or method) to any classifier (with some restrictions specified by OCL).

– DeclareParents. A parent declaration enables adding super classe(s) and/or interface(s) to any classifier (with some restrictions specified by OCL).

– Advice. Advice are dynamic crosscutting features of aspects that affect the behavior of base classifiers. An advice provides a way to express crosscutting action to be performed at the join points that are captured by its attached pointcut. Here is an example of well-formedness rules for *Advice*.

```
context Advice inv: (self.adviceType = #before or self.adviceType = #after)
                    implies (self.parameter->forAll ( p: Parameter | p.kind = pdk_in))
```

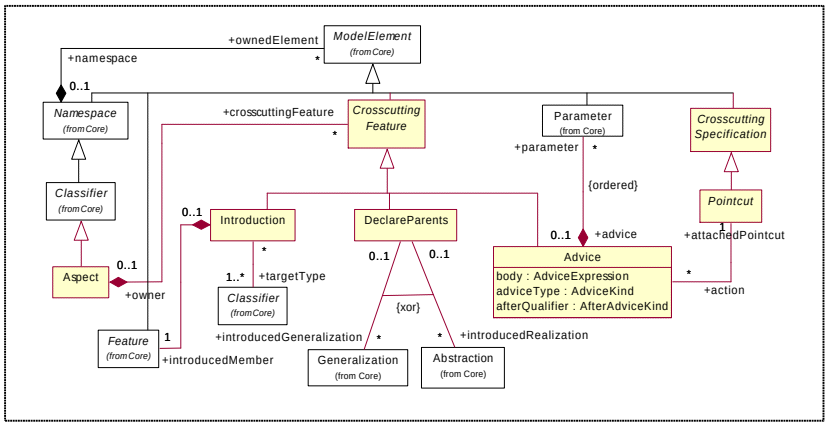


Figure 2. *Abstract Syntax of CrosscuttingFeature*

3.1.3. *Crosscutting specifications*

Crosscutting specifications (Figure 3) specify how static or dynamic crosscuttings have to take place. They define collections of join points (Pointcut) where aspects crosscut classes and/or aspects. In addition, we distinguish other specifications that we consider as being tied to the fact that AspectJ is an extension of Java (CustomCompilation and Soft).

– CustomCompilation. Custom compilation errors or warnings specify messages to be returned at compilation time.

– Soft. Soft declaration allows handling of Java RuntimeExceptions.

– Pointcut. A pointcut identifies a collection of join points that characterizes a dynamic interaction between an aspect and a class. Pointcuts may or not have a name as well as they can have parameters that specify the execution context.

– NamedPointcut. Compared to an anonymous pointcut, a named pointcut has a signature that specifies the name and optionally a set of parameters, in addition of the declaration of its join points. A named pointcut may be abstract.

– PointcutDesignator. Every pointcut is defined in terms of one or more primitive pointcut designators that designate pre-defined sets of join points. A pointcut designator can either be a TypeBasedDesignator and be linked to one or more classifiers, a MethodRelatedDesignator and be linked to one or more methods, or a ControlFlowDesignator and be linked to another pointcut.

– ComposedPointcut. All pointcut designators can be combined using the standard logical operations to produce composed pointcuts. Composed pointcuts can be composed of primitive or composed ones.

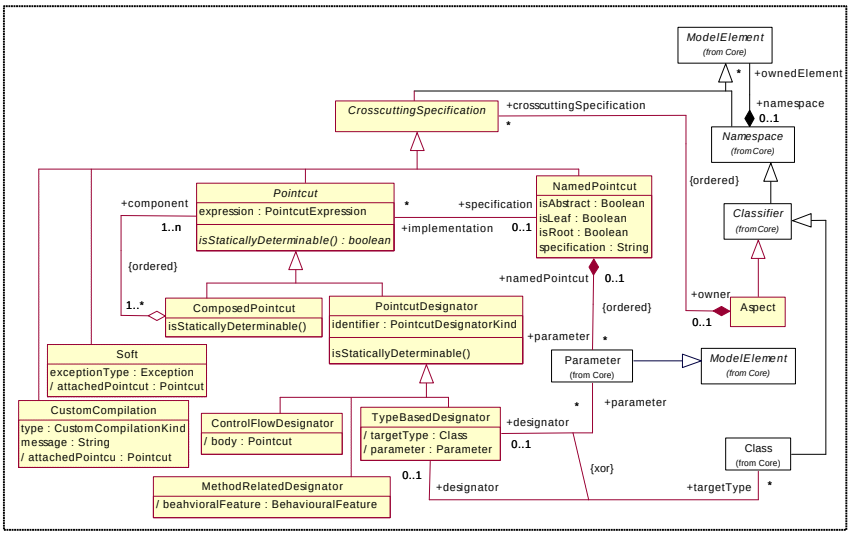


Figure 3. *Abstract Syntax of CrosscuttingSpecification*

3.1.4. *Crosscut and Precedence Relationships*

Each aspect may have precedence relationships with one or more aspects and crosscutting relationships with classes or aspects (Figure 4).

– Precedence. Precedence relationship is defined between two or more aspects, specifying the ordering of advices execution.

– Crosscut. A crosscut relationship is a dependency relationship that relates two or more elements. It states that the implementation or functioning of one crosscutting feature/specification defined in an aspect requires the presence of one or more base classifier (e.g. class, interface or aspect) or features.

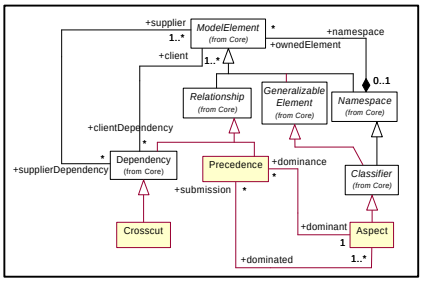


Figure 4. *Abstract Syntax of Crosscut and Precedence*

3.2. *Extending the UML meta-model for Hyper/J: HyperJ/UML*

Hyper/J (Tarr *et al.*, 2000) is an extension of Java for the Hyperspace approach (Ossher *et al.*, 1999). In this subsection we present the UML meta-model extension that we propose to model Hyper/J fundamental concepts.

3.2.1. *Integrable units*

An integrable unit is a model element that may participate in an integration relationship (Figure 5). We distinguish between atomic units (Attribute, Operation, Method) and Composite units (Class, Interface and Hyperslice).

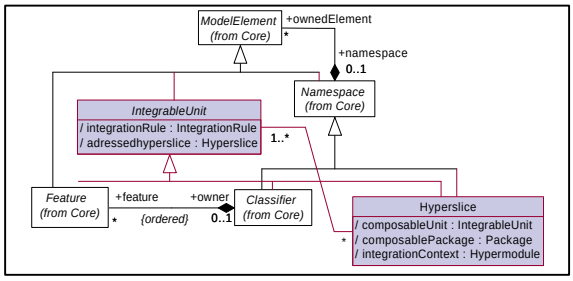


Figure 5. *Abstract Syntax of IntegrableUnit*

3.2.2. *Identification and segregation of concerns*

Hyper/J supports a concern matrix of integrable units, the Hyperspace (Figure 6), which organizes units in a multi-dimensional matrix. Each axis represents a Dimension of concerns and each point on an axis denotes a concern (i.e Hyperslice).

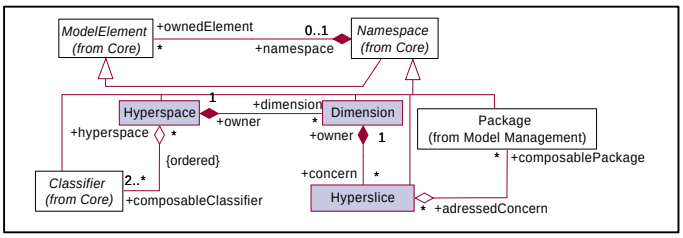


Figure 6. *Abstract Syntax of Hyperspace*

3.2.3. *Composition of concerns: Hypermodule*

Beside the ability to define hyperslices from sets of integrable units, Hyper/J makes it possible to integrate the hyperslices into Hypermodules that provide contexts for a particular software component or system (Figure 7). Each Hypermodule declares the set of hyperslices to be composed and specifies the general composition strategy between their integrable units. There are three kinds of compositionStrategy: *mergeByName*, *nonCorrespondingMerge* and *overrideByName*. The composition strategy may or may not be sufficient to describe the relationships across hyperslices. If needed, definitions of specific integration rules can be added, they are detailed in the next subsection.

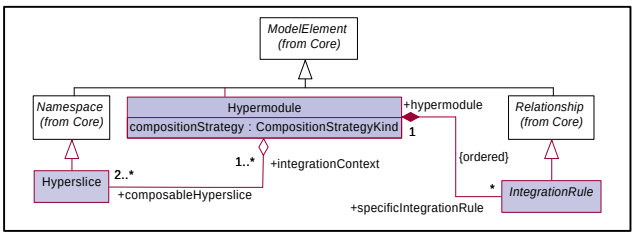


Figure 7. *Abstract Syntax of Hypermodule*

3.2.4. *Specific integration rules*

An IntegrationRule (Figure 8) defines a relationship between one or more integrable units belonging to hyperslices. Specific integration rules are as follows.

– Override and Merge are two specializations of Integration, connecting two or more units and producing primitive or composite integrable units.

– Order and Summary specialize the MergeRule meta-class; they can only be used with a merge relationship for the purpose of solving merge conflicts.

– Rename indicates that a specific unit has to be given a new name.

– Match and Equate are mainly used to indicate that some integrable units are corresponding to each other.

– NoMerge has the opposite effect of the Merge or the Override relationships: it causes two or more units that match each other by name not to be merged or overridden, even if the general integration rule is merge or override by name.

– Bracket indicates if the execution of one or more methods must be preceded and/or followed by the execution of other methods

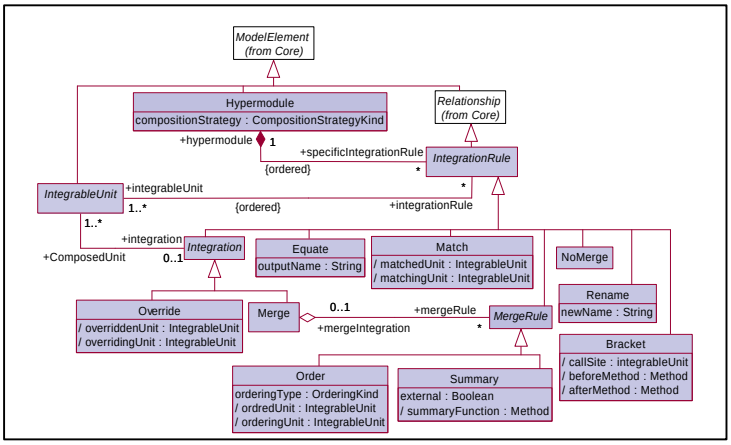


Figure 8. *Abstract Syntax of Integration Rules*

3.3. *Aspect/UML: the aspect-oriented general meta-model*

We present here a general meta-model for aspect-oriented designs. We deduce its elements by comparing the two language-specific meta-models presented above. Indeed similarities between those two specific meta-models are not straightforward: it is quite easy to notice, even if we fragmented both abstract syntaxes into several diagrams that AspectJ/UML mainly extends the UML meta-model on the “concepts side” (see Figure 3) whereas HyperJ/UML mainly consists in the adding of relationships (see Figure 8). We thus did not seek an exhaustive comparison that would certainly put forward too many differences but we rather started an effort to find out correspondences between the fundamental concepts and relationships that we defined in both meta-models. Figure 9 shows the abstract syntax of the fundamental concepts, which are more abstract than the ones denoting the specific concepts of both languages, as well as the way they are related to each other.

– Concern. Concerns are model elements encapsulating base or crosscutting concerns. Concern is an abstract meta-class

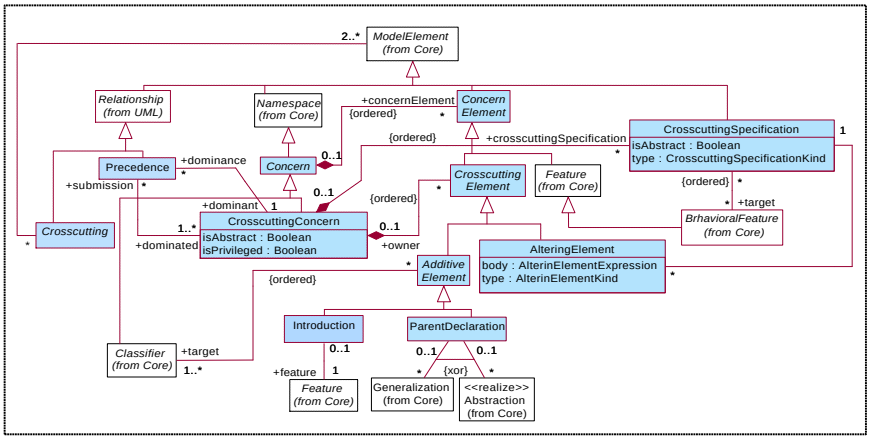


Figure 9. *Abstract Syntax of the fundamental concepts of Aspect/UML*

– *Base Concern* (Classifier). Base concerns are model elements that are affected by one or more crosscutting concerns (Aspects or Hyperslices). As base concerns are classifiers in both languages, we did not add a newly defined meta-class to denote them. Indeed, crosscutting elements in AspectJ and Hyper/J impact either directly or indirectly classifiers through their owned features.

– CrosscuttingConcern. Crosscutting concerns are units encapsulating crosscutting concerns. CrosscuttingConcern is a generalization of the meta-classes that have been introduced in both meta-models for this reason: Aspect in AspectJ/UML and Hyperslice in HyperJ/UML.

– CrosscuttingElement. Crosscutting elements are structural or behavioral properties of a particular crosscutting concern, which affect the base concerns. A Crosscutting element may either be an AlteringElement or an AdditiveElement. Altering elements intend to alter behaviors of base concerns, while additive elements modify their definition by augmenting them. Introduction is an additive element that allows statically adding of some features (structural or behavioral features) into base concerns while keeping their original definitions safe. ParentDeclarations are as well specializations of additive elements. They allow extending base concern definitions by modifying their hierarchies (i.e. by adding new super-classes or interfaces to base concerns). In the case of AspectJ/UML we mainly distinguish between Introduction, DeclareParents and Advice. In HyperJ/UML base Features are crosscutting properties of Hyperslices: a feature is more specific than an integrable unit that can be aggregated in a hyperslice.

– CrosscuttingSpecification. A crosscutting specification indicates where crosscutting (performed by some instance of AlteringElement) has to take place as well as it specifies when it has to be executed in the case of a dynamic crosscutting. Each altering element must be associated to one crosscutting specification while each crosscutting specification may be associated to zero or several behavioral

features. A crosscutting specification is abstract if it is not associated to any behavioral feature. Consequently a crosscutting concern has to be abstract if it contains at least an abstract crosscutting specification. The crosscutting specification type indicates when the crosscutting has to take place and may have several values (call, initialization or staticInitialization). The type of an altering element specifies how the impacted behavioral feature would be altered (before, after, replacement, combination or narrowing). This is defined in Hyper/J by means of general and specific integration rules, while AspectJ corresponding definitions are based on pointcut declaration and type specification.

– Crosscutting is a direct relationship between two or more model elements. In both specific meta-models meta-classes have been introduced for this reason: Crosscut in AspectJ/UML and Integration relationships (merge and override) in HyperJ/UML.

– Precedence specifies the ordering of the execution of several altering elements (owned by different crosscutting concerns) that affect the same base concern in the same place. AspectJ use the Precedence relationship to do this, while Hyper/J uses the Order integration rule. In both cases, the precedence relationship is defined between two crosscutting concerns.

The rest of the language-specific concepts are, at the first glance, rather distant and they cannot be directly compared. We thus retain the abstraction of these six common concepts and relationships in our general meta-model.

4. Model transformations

If being able to express language-indepent aspect-oriented designs is an advantage, we also have to consider the transformation of those designs to become more specific so that they can be easily implemented in an actual aspect-oriented programming language. We propose in this section, model transformations for transposing a model which is an instance of the general meta-model, to an instance of one of the specific meta-models, respectively devoted to AspectJ and Hyper/J. We also detail some of the transformation rules specifying the mapping between these meta-models. The *Strategy* pattern serves as an illustration for these transformations.

4.1. *Overview of the transformation process*

In the line of conduct of MDA, we propose to project aspect-oriented general models to language specific ones. Our model transformation approach (Figure 10) takes one general model (1) as input and produces one or more specific models (3). In order to perform and guide such a transformation, some additional information is required that has to be integrated with the general model. Therefore while performing the mapping between source and target models, our approach also relies on intermediate annotated general models (2). Annotations that are added to models in (1) to obtained models in (2) are tagged values. These tagged values can be set by

default and eventually changed by the user. They denote details that are absent in the general model but are relevant when becoming language specific. For example, when transposing a general model to an AspectJ specific model, one has to assign instantiation types (singleton, perThis,…) for each Aspect. A tagged value named instantiation enrich then the general model, it has singleton as a default value. Transformation rules can then be processed on the intermediate models in (2) to obtain models in (3). Each transformation rule specifies a mapping from a general model element to one or several corresponding model elements defined in the target meta-model. We detail some of these transformation rules in the following subsection. In order to illustrate the transformation rules that we propose, we consider in what flows the transformation of the GoF *Strategy* pattern's general model into AspectJ and Hyper/J specific models.

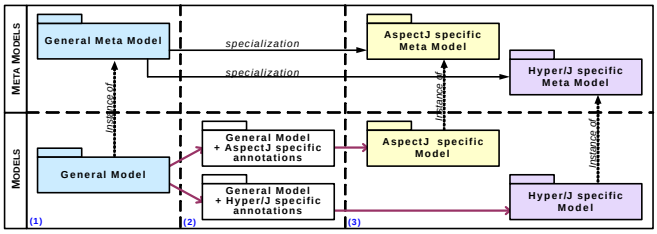


Figure 10. *Transformation process*

4.2. *Transforming the Strategy general model into AspectJ specific model*

We present here some transformation rules (see (Hachani, 2003) for more transformation rules) devoted to the transformation of a general model into an AspectJ-specific one.

- For each CrosscuttingConcern's instance cC_i in the general model i=1..* : create an Aspect's instance a_i in the AspectJ-specific model.

- For each Introduction's instance cCI_j j=1..*, owned by cC_i, create an Introduction's instance aI_j owned by a_i.

- For each Operation's instance cCO_k, k=1..*, owned by cC_i (respectively, each Method's instance cCM_l, l=1..*, associated to cCO_k), create an Operation's instance aO_k (respectively a Method's instance aM_l) owned by a_i (respectively associated to aO_k).

- For each AlteringElement's instance aE_r, r=1..*, owned by cC_i, create an Advice instance aA_r (with the same properties as aE_r) owned by a_i.

- For each CrosscuttingSpecification instance cS_s, s=1..*, owned by cC_i and associated to aE_r, create a Pointcut instance aP_s (having the same properties as cS_s) owned by a_i.

Strategy's intent is to define a family of interchangeable encapsulated algorithms (Gamma *et al*., 1995). It allows giving polymorphic definitions of a method for the instances of the same class. We propose (Table 1(a)) to gather the various definitions of the polymorphic behavior (named defaultAlgorithm(), algorithm1(), algorithm2(),… in the diagram) in one crosscutting concern ContextStrategies for each instance of the *Strategy* pattern. ContextStrategies also introduces an attribute strategy and a parameterized constructor Context(int stg) in the Context class. The strategy attribute is used for the internal representation of the chosen strategy in each instance of Context. A crosscutting specification PerformInterface intercepts all calls to the Context's algorithmInterface() method, in order to replace each of its invocations by the invocation of the appropriate algorithm. An altering element (depicted in our concrete syntax by just its type, replacement in this example) is then defined so that it invokes, in its turn, one of the definitions of the polymorphic behavior, depending on the actual value of the strategy attribute held by the receiver.

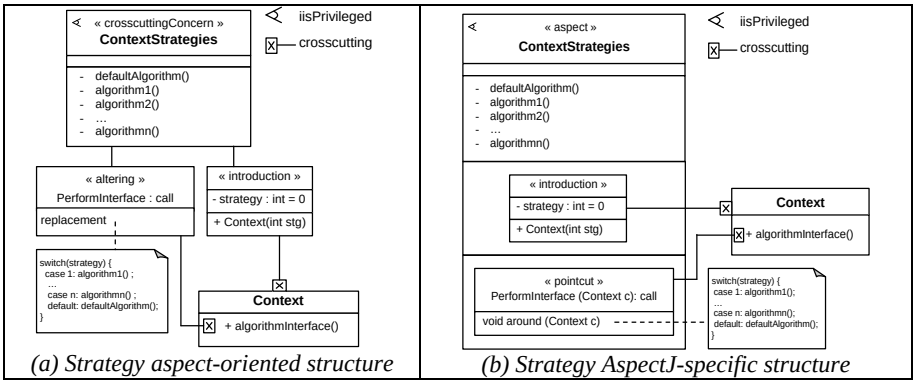


Table 1. *Strategy aspect-oriented structure*

Table1 (b) shows the AspectJ structure of the *Strategy* pattern that is obtained by applying the above transformation rules to the *Strategy* aspect-oriented structure. The CrosscuttingConcern's instance ContextStrategies has been transformed into a privileged Aspect's instance having the same name. The CrosscuttingSpecification's instance became an instance of MethodRelatedDesignator (a specialization of Pointcut), and the AlteringElement's instance has been transformed into an Advice's instance. The value of the adviceType attribute was determined by a transformation rule which specifies that an Advice's instance obtained by transformation of an AlteringElement's instance of type replacement must have an around type.

4.3. *Transforming the* Strategy *general model into Hyper/J specific model*

We now consider the transformation of our *Strategy* general model to obtain a Hyper/J compliant design (Figure 11), by applying the above transformation rules.

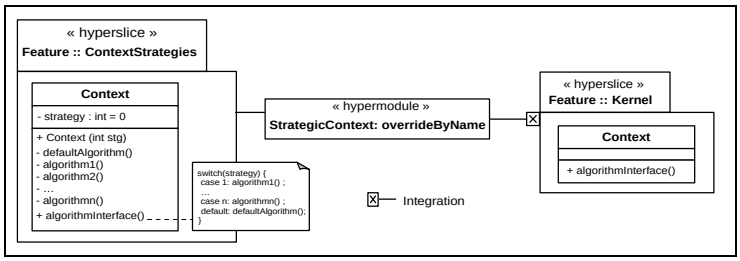


Figure 11. *Strategy Hyper/J-specific structure*

- Create an instance of Hyperspace along with two instances of Dimension: cD for the class dimension and cCD for the crosscutting concern dimension.
- Create an instance of Hyperslice named bH (within the cCD dimension) that contain all base Classifier's instances C_i defined in the general model, i=1..*.
- For each CrosscuttingConcern's instance cC_j, j=1..*, create a Hyperslice's instance h_j within the cCD dimension.
- For each Introduction's instance i_k, k=1..*, owned by cC_j:
  - for each C_i linked to i_k, create a new corresponding instance of Classifier hC_i with the same name as C_i and link hC_i to h_j;
  - for each introduced Attribute's instance iA_l l=1..*, linked to i_k, create as many Attribute's instances hC_iA_l as there are hC_i;
  - for each introduced Operation's instance iO_m, m=1..*, linked to i_k (respectively introduced Method's instance iM_n, n=1..*, linked to iO_m) create as many new Operation's instances hC_iO_m (respectively as many new Method's instances hC_iM_n) as there are hC_i.
- For each AlteringElement's instance aE_p, p=1..*, owned by cC_j as well as for its CrosscuttingSpecification instance cS_p:
  - ensure that for each C_i indirectly specified by the cS_p, a corresponding Classifier's instance hC_i exists within h_j;
  - create as many Operation's instances of hC_iO_p, with the same name as the Operation's instance linked to the cS_p, and whose body hC_iM_p is the same as the body of the corresponding aE_p.
- For each Operation's instance cCO_q, q=1..*, owned by the cC_j (respectively for each Method's instance cCM_r, r=1..*, linked to the cCO_q), create new Operation's instances hC_iO_q owned by hC_i (respectively new Method's instances hC_iM_r).
- Finally create an instance of Hypermodule with (by default) an overrideByName general integration rule. Create the needed IntegrationRule's instances with respect to the cS_p properties.

The CrosscuttingConcern's instance has been transformed into a Hyperslice's instance named Feature::ContextStrategies (where Feature is the name of the dimension and ContextStrategies is the name of the concern). This hyperslice consists of a new Class's instance Context, that includes all added features and the

various definitions of the polymorphic behaviour. The AlteringElement's instance has been transformed into a new interface() method, also owned by the new Context class. Moreover, the base Context class is declared within an additional Hyperslice's instance named Feature::Kernel. It can be noticed then that there is not much information needed to specify how to compose these two hyperslices. The overrideByName general integration rule of the Hypermodule's instance StrategicContext is indeed adequate enough. This rule is moreover sufficient to deal with all introductions, as well as the AlteringElement's instance with type replacement and whose body is specified by the body of the new interface() method.

5. Related works

(Aldawud *et al.*, 2003), (Basch *et al.*, 2003), (Stein, 2003), (Suzuki *et al.*, 1999) propose to extend UML using its inherent extension mechanisms (stereotypes, tagged values and constraints) to obtain a language that is suitable for aspect-oriented modeling. As we briefly discussed it in Section 2, we consider such approaches to be insufficient because the definition of aspect-oriented constructs has then to be too much related to the definition of strictly object-oriented constructs.

Some other works are related to providing a meta-model for either AspectJ (Lions *et al.*, 2002), (Mancona *et al.*, 2002) ((Chavez *et al.*, 2001) does not claim to be related to AspectJ but considers language constructs that are very close) or Hyper/J (Philippow *et al.*, 2003) but none of them do consider a meta-model complying with, or more general than, both languages. The meta-models considered in these works present some similarities with the ones we propose but also some differences: (Philippow *et al.*, 2003) for example defines the Hyperslice meta-class as a specialization of Package whereas we consider a hyperslice to be a namespace which is more general than a package. (Clarke, 2001) propose a language independent meta-model for aspect-orientation, starting from the notion of composition pattern which is more closely related to the Hyper/J than to AspectJ. This meta-model enables projection of models to both AspectJ code and Hyper/J code, these projections are essentially based on translations from one model element to programming language constructs. We chose to adopt another approach by transforming models into language-specific models. We are convinced that one could benefit from such an intermediate stage to do some rework of the designs once the target programming language has been determined.

6. Conclusion and future works

We have presented in this paper an approach that makes it possible to express aspect-oriented designs in a language independent manner, and transpose them into aspect-oriented language-specific designs. This approach is based on an extension of the UML meta-model including concepts that are abstract enough to encompass the features of two of the most today known aspect-oriented programming languages,

AspectJ and Hyper/J. We also provided two other extensions of the UML meta-model that are respectively specific to these two programming languages. Model transformation rules complete the proposition of this approach in order to ensure the transposition of general models into corresponding language-specific ones.

We started this work by referring to UML 1.5 as the current standard specification and stepping our meta-model extensions to UML 2.0 is one of the improvements that we may achieve on this work. Another improvement we are already working on, and that is directly related to the UML extension part of our work, is to finalize the concrete syntaxes associated to the 3 meta-models presented in this paper (interested readers may eventually check the progress of this work in (Hachani, 2003) as well as get more details on the *abstract syntaxes*, *well-formedness rules* and *semantics*). It can be argued that we limited our approach to only two aspect-oriented programming languages whereas much more languages are today proposed to the AOSD community. This does not foster our main motivation which is related to the aspect-oriented evolution of design patterns because no total but rather a representative coverage of programming languages is required for design patterns description. We are however confident in the possible extensions of our work to several other aspect-oriented languages, such as AspectS for instance, since it is based on constructs that are closely related to those of AspectJ. We also hope that our work on the characteristics of both AspectJ and Hyper/J (and even other languages if our approach would ever have to be extended to them) will improve the overall understanding of these two languages and more generally aspect-orientation.

7. References

Aldawud O., Elard T., Bader A. UML profile for Aspect-Oriented Software Development. *AOSD'03 3rd Workshop on Aspect-Oriented Modeling with UML*, 2003.

Basch M., Sanchez A. Incorporating Aspects into the UML. *AOSD'03 3rd Workshop on Aspect-Oriented Modeling with UML*, 2003.

Chavez C., Lucena C. Design–level Support for Aspect-oriented Spftware Development. *OOPSLA 2001 Workshop on Advanced Separation of Concerns*, 2001.

Clarke S. Composition of Object-Oriented Software Design Models. PhD Thesis, Dublin City University, 2001.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Hachani, O., Bardou, D. « Aspectisation » de patrons de conception. In *Actes du XXIème congrès INFORSID 2003* (Nancy, France, 3-6 juin, 2003), 153-168.

Hachani O., Bardou, D. On Aspect-oriented Technology and Object-Oriented Design Patterns. *ECOOP 2003 Workshop on Analysis of Aspect-Oriented Software*, 2003.

Hachani O. Extending UML meta-model for AspectJ, HyperJ and aspect-orientation. Research reports. http://www-lsr.imag.fr/Les.Personnes/Ouafa.Hachani/works.html.

Hannemann, J., Kiczales, G. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 2002 ACM SIGPLAN Conference on OOPSLA 2002*, SIGPLAN Notices, Vol. 37, N°11, ACM (2002), 161–173.

Hirschfeld R. AspectS – Aspect-oriented Programming with Squeak. In *Proceedings of the International Conference NetObjectDays (NODe 2002)*, (Erfurt, Germany, Oct. 2002), LNCS, vol. 2591, Springer, 213-232.

Hirschfeld, R., Lämmel, R., Wagner, M. Design Patterns and Aspects – Modular Designs with Seamless Run-Time Integration. *The 3rd German Workshop on Aspect-Oriented Software Development* (AOSD-GI 2003), 2003.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G. An Overview of AspectJ. In *Proceedings of ECOOP 2001*, LNCS, Vol. 2072, Springer (2001), 327–353.

Lions J.M., Simoneau D., Pilette G., Moussa I. Extending OpenTool/UML Using Metamodeling : An aspect-oriented programming case study. *UML'02 2nd Workshop on Aspect-Oriented Modeling with UML*, 2002.

Mancona Kandé M., Kienzle J., Strohmeier A. From AOP to UML – A Bottom-Up Approach. *AOSD'02 1st Workshop on Aspect-Oriented Modeling with UML*, (Enschede, The Netherlands, April, 2002), 2002.

Noda, N., Kishi, T. Implementing Design Patterns Using Advanced Separation of Concerns. In *OOPSLA 2001 Workshop on AsoC in OOS*, 2001.

Nordberg, M.E. Aspect-Oriented Indirection – Beyond Object-Oriented Design Patterns. *OOPSLA 2001 Workshop "Beyond Design: Patterns (mis)used"*, 2001.

Ossher H., Tarr P.L. Hyper/J™: Multi-dimensional separation of concerns for Java™, In *Proceedings of the ICSE 2000, International Conference on Software Engineering*, Limerick, Ireland, June, 2000.

Ossher H., Tarr P.L. Multi-Dimensional Separation of concerns in Hyperspace. *ECOOP'99 Workshop on Aspect-Oriented Programming*, 1999.

Philippow I., Riebisch M., Boellert K. The Hyper/UML Approach for Feature Based Software Design. *UML'03 4th Workshop on Aspect-Oriented Modeling with UML*, 2003.

Stein D. *An Aspect-Oriented Design Model Based on AspectJ and UML*. Master Thesis, University of Essen, Germany, 2002.

Suzuki, J., Yamamoto, Y. Extending UML with Aspects: Aspect Support in the Design Phase. *ECOOP'99 Workshop on Aspect-Oriented Programming*, 1999.

Tarr P.L., Ossher H. Hyper/J™ User and Installation Manual. http://www.research.ibm.com/hyperspace, 2000.

Wegner, P. Dimensions of object-based language design. In *Proceedings of the 2nd ACM Conference on OOPSLA'87* (Orlando, Florida, October 4-8, 1987). *ACM SIGPLAN Notices, 22*, 12 (Dec. 1987), 168-182.

AspectJ Team, The AspectJ Programming Guide, http://www.eclipse.org/aspectj/, 2002.

Object Management Group (OMG): Unified Modeling Language (UML) Specification 1.4.2, http://www.omg.org/cgi-bin/doc?formal/04-07-02, 2004.

Object Management Group (OMG), UML 2.0 Object Constraint Language (OCL) Specification, http://www.omg.org/cgi-bin/doc?ptc/2003-10-14, 2003.

Actes de l'atelier de travail
Motifs de méta-modélisation

IDM 2006

Ingénierie dirigée par les modèles appliquée à la conception d'un contrôleur de robot de service

Dave Thomas* — Claude Baron* — Bertrand Tondu*

* *Laboratoire d'Étude des Systèmes Informatiques et Automatiques*
INSA-DGEI de Toulouse
135 avenue de Rangueil, F-31077 Toulouse cedex 4
{dave.thomas, claude.baron, bertrand.tondu}@insa-toulouse.fr

RÉSUMÉ. *Les approches basées modèles marquent une réelle démarche d'évolution dans la conception de systèmes au sens large. Les différentes étapes du développement d'un système, jusque là essentiellement supportées par des documents textuels (informels), sont de plus en plus formalisées afin d'éviter les ambiguïtés, garantir leur complétude et leur cohérence et d'assurer au plus tôt l'adéquation de la solution décrite et de l'expression du problème.*

Dans le cadre de travaux sur une robotique de service anthropomorphe, nous définissons et expérimentons une approche de modélisation évolutive pour le développement du logiciel de commande temps-réel d'un prototype de bras à sept axes actionnés par muscles artificiels. A partir du modèle d'analyse orienté métier plusieurs étapes sont définies afin de guider la conception vers un modèle d'implémentation en pratiquant par transformations successives. Les modèles servent alors de référence à la fois pour le logiciel final ainsi que pour les analyses et les vérifications.

ABSTRACT. *Model-based engineering is going to modify the way you design systems. It encourages replacing textual documents with formal representations in order to enhance verification, coherency and requirements fulfilment.*

The purpose of this paper is to make an overview of how we considered the model-based contribution in support to control system engineering, based on the design of a seven-degrees-of-freedom robot-Arm driven by pneumatic artificial muscles for humanoid robots.

MOTS-CLÉS : *MDA, UML, AADL, temps-réel, robotique.*

KEYWORDS: *MDA, UML, AADL, real-time, robotics.*

2ème soumission IDM 2006 – 1er juin 2006

1. Introduction

1.1. *Contexte et besoins*

Par opposition à la robotique industrielle, dont les applications sont essentiellement manufacturières, la robotique de service vise à sortir le robot du cadre contraignant de l'atelier ou de la chaîne de montage. La fédération internationale de robotique (IFR) a suggéré en 1997 la définition suivante : "A service robot is a robot that operates partially or fully autonomously to perform services useful to the well-being ... of humans and equipment. They're mobile or manipulative or a combination of both" (cité dans Service Robot [Schraft, 2000], page 2). La robotique de service couvre donc un très large champ d'application, du BTP à l'agriculture, en passant par la surveillance, le marketing, les loisirs, la médecine et l'assistance aux personnes. La sécurité active et la fiabilité sont, par conséquent, des impératifs fondamentaux de la mise en œuvre des robots de service.

L'équipe de robotique du LESIA développe depuis plusieurs années une plate-forme expérimentale définie autour d'un bras-robot anthropomorphe à 7 degrés de liberté actionnés par muscles artificiels de McKibben ([Tondu, 2005]). La mise en place d'une telle plateforme nécessite les compétences de plusieurs domaines d'ingénierie tels que la robotique, la mécanique, l'automatique ou l'informatique.

Cette plateforme inclut un système informatique complexe. Ce dernier est constitué d'un logiciel temps-réel, appelé contrôleur de robot, et de son support d'exécution reposant sur un exécutif temps-réel, dans notre cas VxWorks. Le contrôleur de robot héberge les lois de commande et les algorithmes de dynamique et de cinématique élaborés par les roboticiens et les automaticiens. L'approche traditionnelle de développement, organisée autour d'un codage à la main et d'une implémentation mono-tâche, a très vite montré ses limites vis-à-vis de la complexité de la plateforme à 7 axes et nous avons mis en évidence la nécessité d'une démarche guidant le concepteur. Sans cette méthodologie appropriée, il devient très difficile de réaliser le logiciel qui doit être à la fois flexible pour s'adapter aux différents prototypes et sûr de fonctionnement, afin de satisfaire les contraintes de sécurité.

1.2. *Évolution des réflexions*

Nos travaux se sont tout d'abord orientés vers l'emploi des méthodes SADT et SART ([Carroll, 1999]) pour la conception fonctionnelle et temps-réel de nos contrôleurs de robot. Les résultats ont montré des avantages notables dans la maîtrise du logiciel. Pourtant, pour les roboticiens, un certain nombre de schémas utilisés dans ces méthodes restaient difficilement projetables sur la plateforme réelle et un manque de lisibilité subsistait. L'approche objet présentait l'avantage de réconcilier le domaine du logiciel avec le monde réel en basant les démarches sur la représentation des éléments concrets (les objets). Aussi, une modélisation à l'aide du langage UML du contrôleur a été réalisée, permettant d'améliorer la compréhension du logiciel ([Guiochet, 1999]). Ensuite, le modèle a été remanié dans le but d'accroître la flexibilité et la possibilité de réutilisation de certaines parties

(composants) du logiciel ([Thomas, 2004]). Une architecture de référence a alors été proposée afin d'abstraire les parties spécifiques de bas-niveau pour concentrer les efforts sur la partie fonctionnelle.

1.3. *Les approches MDA et MDE*

Séparer explicitement solution fonctionnelle (logique) et solution technologique (physique) est nécessaire pour faciliter le déploiement d'un logiciel sur différentes plateformes d'exécution. L'approche MDA (Model Driven Architecture [OMG, 2003]) propose à ce sujet de s'appuyer sur les modèles pour supporter les deux types de représentations. Des liens peuvent ensuite être tissés entre les modèles pour permettre de les assembler de manière automatisée et produire le logiciel final.

L'ingénierie dirigée par les modèles (MDE, Model Driven Engineering) est la généralisation de l'approche MDA [Favre et al., 2006]. Les avantages annoncés de l'IDM sont nombreux : indépendance vis à vis des évolutions technologiques, meilleure maîtrise de la complexité, meilleure réutilisation etc. Le nombre de vues augmentant, les modèles utilisés et leur sémantique associée sont de mieux en mieux définis et, petit à petit, les modèles occupent ainsi une position centrale dans le processus de développement, celle de prototype virtuel du système à réaliser. En effet, l'évolution des ateliers logiciels (tels que I-Logix™ Rapsody, Telelogic™ Tau, Artisan Software™ Studio) et des langages autorise aujourd'hui la construction de modèles exécutables (cf. Executable Systems Design with UML2.0 [Niemann, 2004]). L'analyse et la simulation sont alors permises et facilitent la correction au plus tôt d'erreurs détectées jusqu'à maintenant seulement pendant les phases de tests, après codage. L'utilisation d'une représentation simulable (le modèle) peut ainsi réduire significativement les temps et les coûts des itérations de maintenance corrective (débogage). Enfin, l'approche généralise également la transformation de modèles. En effet, que ce soit pour appliquer un patron de conception, pour séparer les préoccupations à travers différentes vues ou pour réaliser des fusions, les transformations seront utilisées pour modifier, créer ou même supprimer certains éléments du modèle. De même, la génération de code ou de documents constitue un autre type de transformation (modèle vers texte) qui oriente les modèles vers une utilisation plus productive. Plus généralement, la transformation de modèles servira de moyen pour automatiser certaines tâches ou, du moins, les accélérer.

Cet article présente une mise en œuvre de ces concepts à travers la conception du contrôleur de robot à 7 axes présenté paragraphe 1.1.

La première partie, section 2, est consacrée à la démarche de modélisation qui, faisant référence au MDA, mène à construire tout d'abord une architecture fonctionnelle indépendante de la plateforme puis l'architecture dynamique et physique sur laquelle elle repose.

La deuxième partie, section 3, s'intéresse alors à la vérification et à la validation du logiciel à partir de ces modèles.

Enfin, la troisième partie a pour but de résumer la démarche suivie et de faire la synthèse sur les concepts employés et les leçons apprises.

2. Modélisation du contrôleur de robot

La modélisation du contrôleur de robot consiste à définir l'architecture du logiciel de contrôle du robot ainsi que les interfaces nécessaires en vue de sa projection sur son support d'exécution. Cette section présente tout d'abord la phase d'analyse des besoins depuis laquelle nous sommes partis pour construire les modèles. Les éléments de l'architecture fonctionnelle en sont déduits et dérivés progressivement. Enfin, la prise en compte des contraintes d'implémentation est abordée après avoir modélisé le comportement à l'exécution en décrivant l'architecture dynamique

2.1. *Analyse des besoins*

Dans le but de supporter la phase de capture des besoins, nous avons fait le choix d'une approche aujourd'hui devenue classique, basée sur le langage UML. Les cas d'utilisation décrivent les besoins selon le point de vue de l'utilisateur et les illustrent de manière graphique à l'aide de diagrammes. Cet apport visuel favorise les discussions avec le client, permet de situer facilement le périmètre du système ainsi que ses interfaces avec le monde extérieur. Ensuite, la description des scénarios est réalisée à l'aide de diagrammes de séquences à travers lesquels sont analysés les interactions de haut niveau et les objets primaires du système.

A ce premier niveau d'analyse, une architecture statique d'objets se dessine. La spécification peut être complétée d'une description comportementale sous forme de machines à états associées aux objets. On obtient alors un **modèle d'analyse** (cf. Figure 1.a) composé des objets représentatifs du système et de leurs associations.

2.2. *Modèle indépendant de la plateforme*

2.2.1 Architecture statique

Après avoir traité les fonctionnalités du système, le concepteur s'efforce de faire évoluer l'architecture afin de répondre à des besoins moins fonctionnels et tendre vers un modèle structurel d'implémentation en décomposant les objets. Par exemple, la plateforme robotique est avant tout un environnement d'expérimentation qui doit faciliter l'introduction de nouveaux algorithmes et de nouveaux prototypes de bras ou, plus généralement, de systèmes articulés. La flexibilité et la réutilisabilité sont des propriétés recherchées dans le but de mettre en œuvre un contrôleur générique. Notre démarche consiste à illustrer le travail du concepteur à la fois pour capturer son savoir-faire et pour identifier ses besoins. Par exemple, pour chaque lien qui existait entre deux objets une interface a été créée pour réduire le couplage (cf Figure 1.b), ce qui pourrait facilement être automatisé au niveau outil à l'aide d'une transformation de modèles.

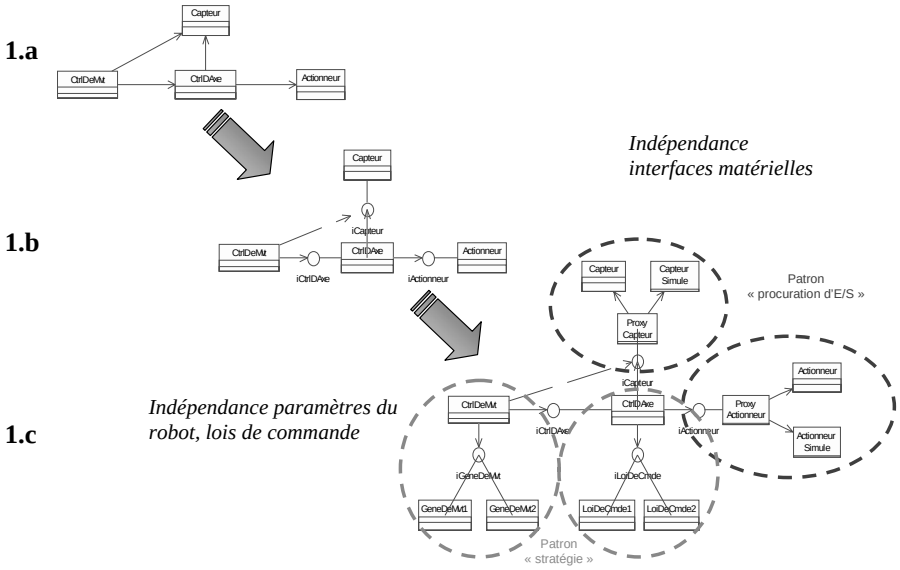


Figure 1. *Construction de l'architecture fonctionnelle par transformation*

2.2.1.1 *Patrons de conception*

Ensuite, Figure 1.c, nous nous sommes inspirés des Patrons de Conception (design patterns) dont il existe aujourd'hui de nombreux modèles regroupés en catalogues, le plus connu étant celui du GoF ("Gang of Four", auteurs de [Gamma et al., 1999]).

Par exemple, le patron « procuration d'entrée/sortie » (ou proxy) a été appliqué aux objets capteurs et actionneurs qui représentent les liens physiques avec le robot. Ce patron crée un mandataire à la place du capteur qui va nous permettre, selon le mode choisi, de relier les algorithmes de commande soit au robot physique, soit à un modèle simulé du robot (maquette java3D) sans avoir à modifier tous les objets qui utilisent l'objet capteur. D'autres fonctions sont permises par l'utilisation d'un mandataire comme le traitement du signal (le mandataire filtre les données du capteur) ou la protection contre les accès concurrents (cf. 2.2.2.2).

Le patron « stratégie » a également été appliqué pour séparer la partie contrôle et la partie purement algorithmique. Les classes qui implémentent les lois d'asservissement ou de génération de mouvement comprennent par exemple plusieurs méthodes et attributs liés à leur exécution (lancement de la tâche et périodicité par exemple) et à la communication, indépendamment de la méthode réalisant le calcul. Le patron stratégie « définit une famille d'algorithmes, encapsule chacun d'entre eux et les rend interchangeables. Le modèle Stratégie permet aux

algorithmes d'évoluer indépendamment des clients qui les utilisent. » [Gamma et al., 1999]. De plus, cette solution architecturale permet de disposer de plusieurs algorithmes et de choisir dynamiquement celui qui convient le mieux selon la tâche à effectuer.

2.2.1.2 *Cadre d'application*

L'ajout de ces points de connexion nous a permis de séparer, d'un coté, ce qui constituait le cœur fonctionnel du contrôleur de robot, et d'un autre coté, les objets et le code plus spécialisés concernant les algorithmes de commande et les interfaces avec le système articulé (cf. Figure 2). Tous ces concepts nous ont ainsi amené à définir ce que l'on nomme un cadre d'application (framework), un ensemble de classes abstraites dont les interactions sont définies et avec lequel le développeur peut créer simplement des applications spécifiques en injectant du code spécialisé.

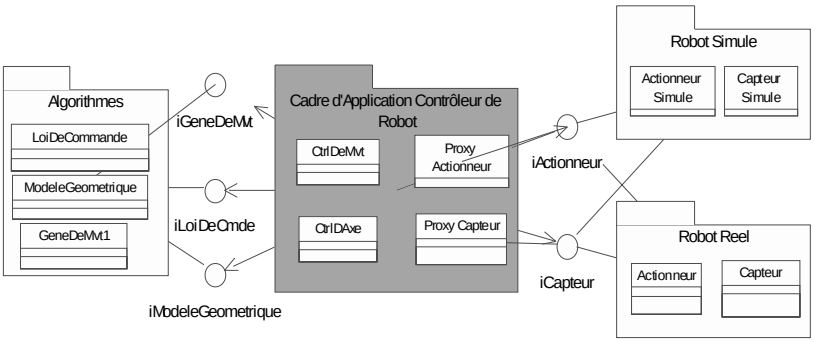


Figure 2. *Cadre d'application "Contrôleur de Robot" (vue simplifiée)*

2.2.2 Architecture dynamique

Le comportement temporel du contrôleur et surtout sa prédictibilité font partie des propriétés essentielles à vérifier dans le cadre des systèmes temps-réels. Il faut d'abord identifier les ressources nécessaires pour l'exécution du logiciel (tâches) puis mettre en œuvre les mécanismes de synchronisation et de communication.

2.2.2.1 *Expression de la concurrence*

Dans l'approche objet, chaque objet a, par construction, une structure propre, et habituellement, on lui associe une machine à états décrivant son comportement. Par conséquent, tous les objets sont potentiellement en concurrence. Faut-il alors associer une ressource d'exécution à chaque objet ? Heureusement non, et il est nécessaire d'identifier clairement quelles sont les sources de comportement concurrent, ce qui n'est pas toujours facile. De plus, en pratique, un objet pourra nécessiter aucune, une ou même plusieurs ressources d'exécution concurrentes.

Cette concurrence est exprimée de plusieurs façons :

– explicitement : en UML, une classe sera soit active soit passive. Chaque instance d'une classe active est associée à au moins un flot d'exécution propre. Les autres objets, dit passifs, ne sont essentiellement utilisés que pour la manipulation de données et s'exécutent sous le contrôle des objets actifs qui interagissent avec eux ;

– implicitement dans le cas où, par exemple, les machines à états font référence à des états concurrents (composites ou imbriqués) dans lesquels plusieurs activités sont suivies au même moment.

Au niveau implantation, sur une plateforme monoprocesseur, une ressource d'exécution sera représentée par une tâche qui correspond à une exécution séquentielle de fonctions (une méthode d'un objet par exemple). Par contre, il est difficile de connaître a priori, par exemple, dans quel flot d'exécution (au sein de quelle tâche) vont s'exécuter les activités associées aux autres méthodes. Dans le paragraphe suivant, nous tentons de prendre en considération les différents cas qui peuvent se présenter.

2.2.2.2 *Communication entre objets*

Dans l'approche objet, les communications se font par échanges de messages, habituellement représentés par des appels de méthodes.

Considérant un cas simple de communication d'un objet actif A vers un objet B, si tout d'abord, l'objet B est passif, la communication sera généralement de type synchrone, c'est-à-dire que l'objet A appelle une méthode de l'objet B et l'exécute au sein de son flot d'exécution (cf. 2.2.2.1). Si l'objet A est le seul à accéder à l'objet B qui ne communique pas avec d'autres objets, aucun problème de concurrence ne peut apparaître. Par contre, si l'objet B est accédé depuis plusieurs objets actifs (ie. plusieurs tâches) de manière asynchrone ou si il est lui-même actif, son intégrité est à contrôler. En effet, tout attribut d'un objet, modifiable à l'aide d'une méthode de celui-ci, devient une ressource critique. En UML, à travers l'attribut *concurrency*, on peut indiquer pour chaque opération si elle peut être utilisée de manière concurrente ou non. Il faudra alors peut-être interdire l'accès concurrent à l'aide de mécanismes d'exclusion mutuelle par exemple ou réaliser un appel asynchrone. UML2.0 a d'ailleurs introduit la notion de signaux qui sont traités de cette manière par envoi / réception d'évènements. Ainsi, une approche générique d'implémentation de ces objets consiste à leur associer une file d'évènements et chaque appel de méthode publique ne fait qu'y déposer un message. La tâche associée réceptionne alors les requêtes de manière séquentielle lorsqu'elle l'a prévu.

2.2.2.3 *Indépendance vis-à-vis de la plateforme*

Dans les langages de haut niveau tels que Java ou Ada, il est possible d'indiquer si une méthode sera protégée ou non contre les accès concurrents. Avec d'autres, comme dans notre cas, l'utilisation du langage C++ et de l'exécutif VxWorks, ces mécanismes nécessitent l'appel de primitives spécifiques fournies par l'exécutif.

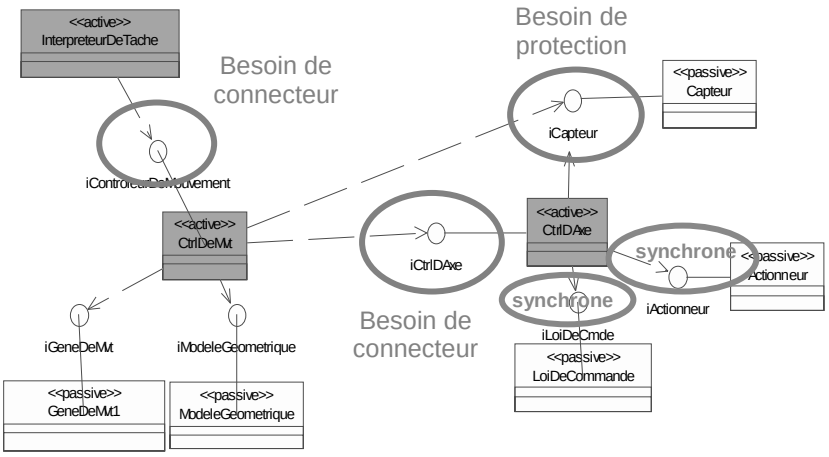


Figure 3. *Architecture dynamique et représentation des besoins*

Par conséquent, si nous modifions dès le modèle afin d'y introduire des éléments spécifiques à notre plateforme, il en deviendra dépendant. Pour rendre le modèle indépendant de cette plateforme tout en explicitant les besoins (voir Figure 3), plusieurs solutions sont envisageables.

La première inclut l'utilisation d'une API[1] standard comme Posix ou Apex ou d'une définie par le concepteur. Changer de plateforme (de système d'exploitation par exemple), supposera alors de redéfinir une implémentation de cette API.

La seconde consiste à modéliser, d'un coté, les besoins du logiciel (en ajoutant des attributs au modèle : tagged values), et d'un autre, les mécanismes fournis par la plateforme (sous forme de patterns) ainsi que les besoins qu'ils couvrent. Le choix est alors laissé au concepteur pour sélectionner et associer les modèles. Par exemple, il est possible de considérer trois types de communication : synchronisation (envoi / réception d'évènement), échange de donnée et transfert de donnée. Une synchronisation peut être implémentée en Posix en utilisant un événement, un sémaphore binaire ou plus simplement par polling en scrutant une variable booléenne. Au concepteur de choisir le mécanisme qu'il souhaite, et d'appliquer le pattern correspondant (qui va créer par exemple les objets et les appels de primitives nécessaires). Cette solution présente l'avantage d'utiliser les fonctions natives de l'exécutif (meilleures performances). De plus, des règles pourraient être vérifiées pour contrôler et valider les choix du concepteur.

Finalement, les deux solutions présentées rejoignent l'idée d'une bibliothèque générique (ou API), de fonctions pour la première et de patterns pour la seconde plus en accord avec l'approche MDA. Dès lors, nous avons entrepris la séparation et

[1] Applications Programming Interface

la modélisation des éléments spécifiques à la plateforme afin de mieux maîtriser le comportement temporel du contrôleur suite à des évolutions architecturales. Il s'agira ensuite de bien identifier quel composant de la plateforme utiliser pour remplir les besoins (de communication, d'exécution) de l'architecture dynamique.

2.3. *Modèle de la plateforme*

Dans cette partie il s'agit de modéliser les services spécifiques à la plateforme qui sont utilisés pour remplir les besoins de l'architecture dynamique (cf. 2.2.2).

Étant donné que l'environnement de modélisation utilisé n'intègre pas encore de solution de transformation de modèles à modèles facilement configurable les modèles ont été réalisés en vue de générer du code directement à travers une bibliothèque d'objets des mécanismes offerts par l'exécutif.

2.3.1 Bibliothèque temps-réel

A partir des librairies écrites en C fournies avec l'exécutif temps-réel VxWorks nous avons créé une bibliothèque de classes permettant d'utiliser de manière simplifiée les mécanismes offerts par le système temps-réel (cf. Figure 4) : tâche, événement (utilisant un sémaphore binaire), sémaphore d'exclusion mutuelle, chien de garde, activations périodiques et gestion du temps.

Par exemple, pour répondre aux besoins des objets actifs, il est nécessaire de pouvoir créer des tâches périodiques ou apériodiques (déclenchées sur événements), ce que permettent les classes de base Processus, ProcessusPeriodique et ProcessusDeclenche (cf. Figure 4). Chaque objet actif hérite de l'une de ces classes selon le type d'activation de l'objet. A l'instanciation de l'objet, une tâche sera lancée, exécutant une méthode virtuelle redéfinie dans la sous-classe correspondante à l'objet actif.

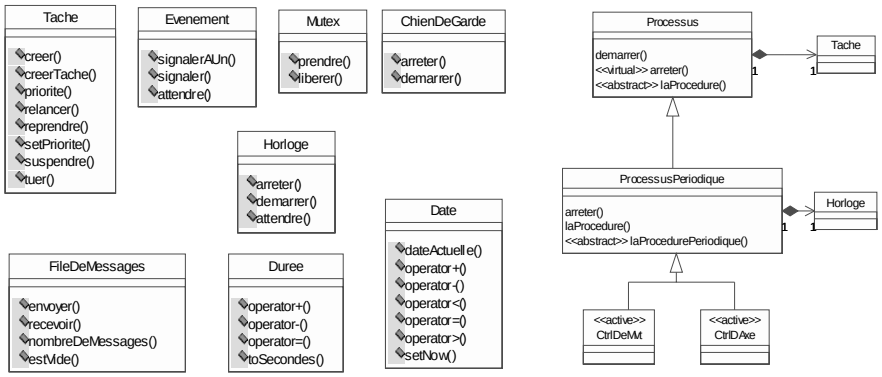


Figure 4. *Bibliothèques d'objets VxWorks*

2.3.2 Connecteurs : éléments de communication

Afin de gérer les flots de données entre objets actifs et au lieu de protéger l'accès aux méthodes des objets actifs, nous avons préféré mettre en place un objet partagé ayant une interface générique, la classe ElementDeCommunication (Figure 5) utilisée comme classe d'association. Deux méthodes, lire() et ecrire() permettent de mettre à jour ou de récupérer la valeur de cette variable partagée. Ensuite, selon les besoins, le concepteur choisit parmi les objets offerts de son exécutif. Par exemple, la classe VariableProtegee, qui implémente cette interface, associe un mécanisme de protection des données (sémaphore d'exclusion mutuelle) au travers de ces méthodes, garantissant qu'aucun processus ne pourra passer outre et la classe BoiteAuxLettres utilise une file de messages.

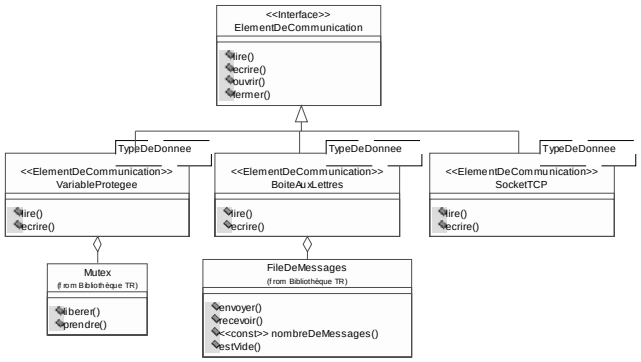


Figure 5. *Classe d'association ElementDeCommunication*

2.4. *Modèle final*

Le modèle d'implémentation final est déduit des deux modèles précédents (cf. 2.2 et 2.3). Les éléments du modèle de la plateforme sont utilisés pour répondre aux besoins identifiés sur le modèle indépendant de la plateforme (cf. Figure 3). Dans notre cas, nous nous sommes limités aux besoins concernant l'exécution temps-réel du logiciel, à savoir l'exécution concurrente, la communication et la synchronisation. Par exemple, les classes actives héritent de la classe Processus afin de leur associer une tâche au sens de l'exécutif choisi (VxWorks). Concernant le besoin de protection identifié pour la classe Capteur, un nouveau mandataire est utilisé mettant en œuvre un sémaphore d'exclusion mutuelle pour conditionner l'accès à l'objet Capteur. Les communications de données, précédemment réalisées par appel de méthode, se font maintenant à travers des objets de la classe ElementDeCommunication (cf. 2.3.2). Il s'agit plus que d'une simple association de modèles et l'obtention du modèle final nécessite l'application de nouveaux patrons de conception réalisant la fusion entre les modèles. La transformation de modèles représente donc réellement un point important dans ce type de démarche.

3. Vérification et validation

Comme nous l'avons énoncé en introduction les modèles peuvent servir de support pour les activités de vérification et de validation du système (logiciel dans notre cas).

Certaines analyses peuvent être conduites directement en utilisant les modèles précédents (cf. section 2). Par exemple, des simulations comportementales, à travers l'animation des machines à états et des interactions entre les objets, sont utilisées pour vérifier les scénarios attendus.

D'autres analyses nécessitent l'extraction ou la transformation des modèles réalisés dans un formalisme adapté à un type de vérifications désirées. Par exemple, il serait avantageux de pouvoir déduire un réseau de Petri global modélisant le comportement du système afin de mettre en œuvre les outils de preuve formelle associés.

Dans cette section, nous présentons un exemple portant sur l'analyse d'ordonnancement de l'architecture dynamique. Nous avons pour cela préféré le langage AADL ([SAE, 2004]) au langage UML. En effet, ce nouveau langage offre une syntaxe et une sémantique permettant d'exprimer formellement le comportement temporel des objets : temps d'exécution des tâches, périodes d'activation, ressources partagées et algorithmes d'ordonnancement utilisés, etc...

3.1. *Présentation du langage AADL*

AADL est un langage émergeant développé sous l'autorité de la SAE (Society of Automotive Engineers) pour répondre aux besoins spéciaux des systèmes embarqués temps-réel critiques tels que les systèmes avioniques. En particulier, le langage peut décrire des vues fonctionnelles de composants échangeant des flots de contrôles ou de données, et leur associer des aspects non-fonctionnels précis tels que les exigences temporelles, les modèles de fautes et d'erreurs, le partitionnement temporel et spatial et les propriétés de sûreté et de certification.

La description d'une architecture en AADL consiste essentiellement à la représentation d'une architecture dynamique de logiciel associée et liée à une plateforme d'exécution. En pratique, elle se décrit à l'aide de composants et de leur composition. Cette composition est :

- hiérarchisée, les composants peuvent contenir d'autres composants et créer des sous-systèmes représentables sur une arborescence ;

- interconnectée, les composants communiquent entre eux par des liens fonctionnels (flots de données et de contrôles) et par des liens physiques (bus, mémoire, réseau, …) ;

Dix catégories de composants ont été définies dans le standard selon trois groupes :

- composants logiciels : data, subprogram, thread, thread group et process;

- composants matériels : processor, memory, bus, device ;

- composants système : system (élément composite pouvant contenir des éléments des deux groupes précédents).

Les caractéristiques non fonctionnelles, c'est-à-dire qui n'apparaissent pas directement au niveau du code, sont modélisées à l'aide de propriétés assez proches des tagged values d'UML2.0. Elles peuvent exprimer par exemple des contraintes, ou des hypothèses qui garantissent la cohérence des composants.

Dans le domaine des systèmes embarqués, certaines propriétés non fonctionnelles sont traitées comme des propriétés fonctionnelles importantes, déterminantes même, pour choisir une architecture ; c'est notamment le cas des propriétés temporelles. Le standard AADL prédéfinit plusieurs propriétés de ce type (cf. Figure 6). Le type et les unités des données peuvent être clairement définis.

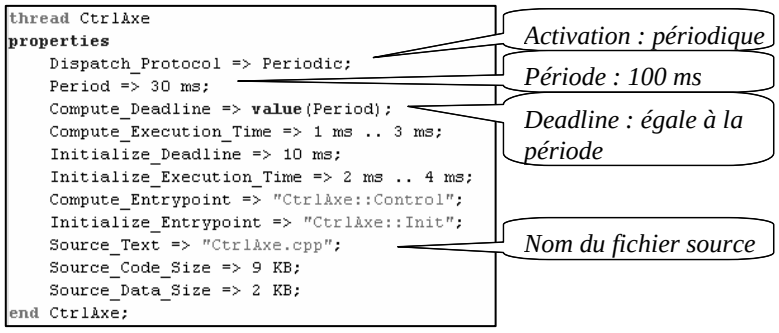


Figure 6. *Exemple de propriétés associées à un thread en AADL (cf. 3.2)*

3.2. *Transformation des modèles UML en AADL*

Afin de réaliser des analyses d'ordonnancement, nous nous limitons aux composants AADL suivant :

- process : représente un espace d'adressage virtuel ; il contient le code binaire d'une application et peut être protégé à l'exécution contre des écritures ou des lectures non autorisées ;

- thread : représente un flot d'exécution parallèle ;

- data : représentent les variables statiques présentes dans le code source. Elles peuvent être partagées entre les threads ou les process et des propriétés (cf.1.4.6) sont utilisées pour définir le type de mécanisme employé pour gérer les accès concurrents ;

- processor : représente une abstraction du matériel et du logiciel responsable de l'ordonnancement et de l'exécution des threads.

Ainsi, à partir de l'architecture dynamique définie au chapitre 2, le modèle AADL (cf. Figure 7) a été construit selon un procédé automatisable, en associant par exemple un composant thread à chaque objet actif et un composant data à chaque ElementDeCommunication partagé. La cible temps-réel est composée d'un seul processeur relié aux capteurs et aux actionneurs par l'intermédiaire de deux cartes d'acquisitions.

Les valeurs des propriétés concernant le type d'activation (périodique, sporadique, apériodique, background, …), les périodes, les échéances et les temps d'exécution ont ensuite été rajoutées puisqu'elles n'apparaissaient pas dans le modèle UML.

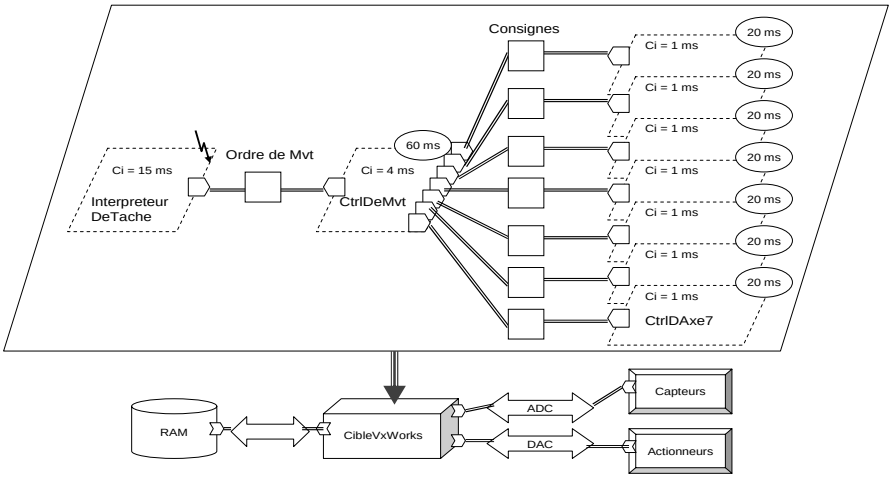


Figure 7. *Architecture dynamique décrite en AADL*

3.3. *Illustration : analyse d'ordonnancement*

La sémantique du langage AADL et les informations formalisées au sein du modèle peuvent ensuite être exploitées par des outils d'analyses, de simulation et de vérification. A l'heure actuelle, ce genre d'outils est encore rare. L'université de Brest a cependant adapté l'outil Cheddar ([Singhoff et al., 2004]), simulateur d'ordonnancement. Nous avons ainsi pu vérifier différents algorithmes et les simuler. Pour illustration, sur la Figure 8, l'exécution temporelle du contrôleur de robot a été estimée en appliquant un ordonnancement de type rate monotonic que nous avons pu comparer aux traces mesurées après codage sous l'outil WindView pour VxWorks.

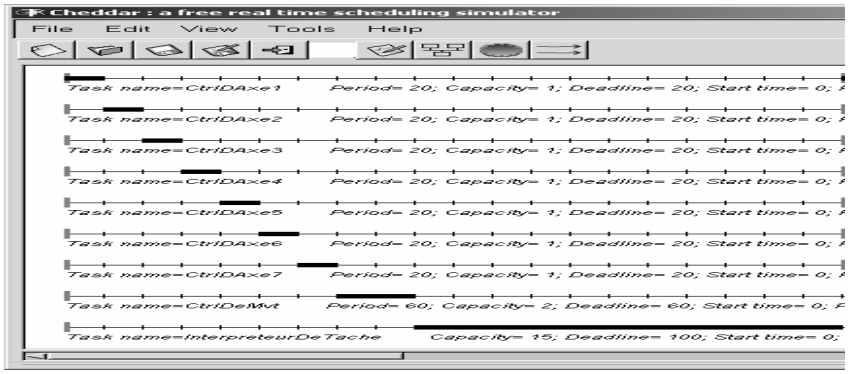


Figure 8. *Exemple de simulation d'ordonnancement avec l'outil Cheddar*

4. Résumé de la démarche

Les approches basées modèles nécessitent avant tout un processus de développement bien défini et une méthode rigoureuse. Avec le souci de mettre en œuvre une méthodologie permettant de rendre la conception et l'implémentation des logiciels de contrôle de robots les plus transparentes possibles, nous avons élaboré une démarche progressive permettant de favoriser la transformation des besoins utilisateurs en une solution concrète.

Ainsi, à partir d'une analyse des besoins guidée par les cas d'utilisation, il nous a paru assez facile d'identifier un premier modèle objet de haut niveau que nous avons appelé **modèle d'analyse** (cf. 2.1), qui correspond au modèle CIM[2] de la démarche MDA [OMG, 2003]. Celui-ci pourra servir de référence à la compréhension des fonctionnalités du logiciel.

Ensuite, nous avons montré que la transformation de ce modèle orienté métier vers un modèle structurel d'implantation pouvait être réalisé par itérations successives, chacune orientée vers l'apport de solutions améliorant la qualité du logiciel en vue de sa réutilisation par exemple (cf. 2.2.1). L'étape suivante a alors consisté à définir les contraintes dynamiques du système en identifiant d'abord les objets actifs puis les besoins de synchronisation et de communication qui en découlent. Cette étape fige le **modèle d'implémentation indépendant de la plateforme (PIM[3])**.

La prise en compte du support d'exécution et de l'exécutif temps-réel utilisés a servi à définir une bibliothèque d'**objets de la plateforme (PDM[4])**. Le **modèle**

[2] CIM = Computation Independent Models (cf. [OMG, 2003])
[3] PIM = Platform Independent Models (cf. [OMG, 2003])
[4] PDM = Platform Description Models (cf. [OMG, 2003])

d'implémentation spécifique à la plateforme (PSM[5]) est le résultat de la transformation du modèle d'implémentation indépendant de la plateforme après que l'on ait sélectionné les objets de la plateforme répondant aux besoins identifiés (concurrence, synchronisation, communication).

Enfin, nous avons apporté l'exemple d'une autre transformation dans laquelle un changement de formalisme était effectué afin de pratiquer des analyses spécifiques d'ordonnancement sur le modèle final.

Cette démarche, présentée Figure 9, a été illustrée dans le cadre de la conception du contrôleur d'un prototype de robot.

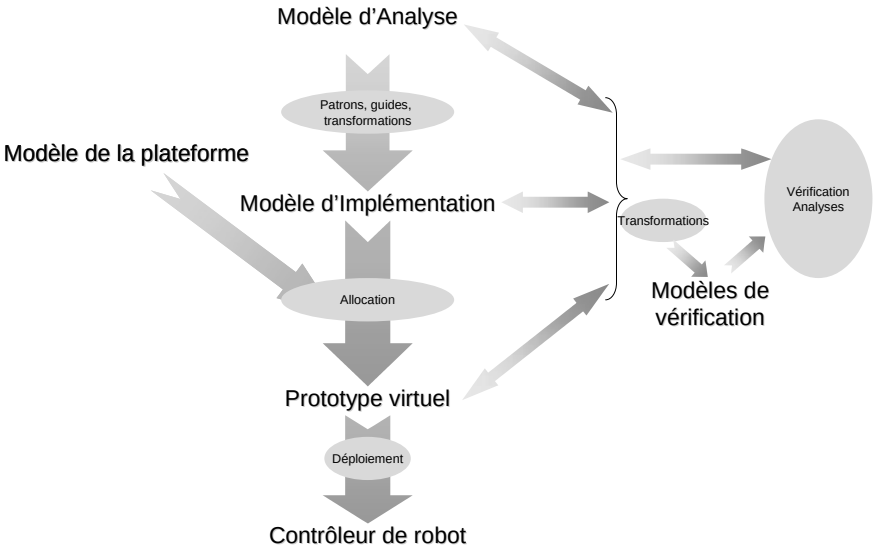


Figure 9. *Démarche de conception appliquée au contrôleur de robot*

5. Conclusion

Dans le domaine de la robotique de service, la satisfaction des besoins de sécurité essentiels sont avant tout dépendants de la parfaite maîtrise du robot. Nous nous efforçons de mettre en œuvre des mécanismes de traitement et de tolérance aux fautes orientés vers la sécurité du personnel. Or, le logiciel de contrôle du robot a une place primordiale dans la réalisation de ces fonctions. L'approche objet nous avait déjà permis d'améliorer la modularité, la flexibilité et l'indépendance vis-à-vis des algorithmes et des interfaces matérielles du contrôleur de robot. La modélisation de la plateforme apporte un meilleur contrôle de son fonctionnement grâce à la prise en compte explicite, au niveau modèle, des problèmes d'implantation. Les modèles

[5] PSM = Platform Specific Models (cf. [OMG, 2003])

représentent alors un support efficace à ces approches et leur utilisation tout au long du cycle de développement, de l'analyse des besoins à l'implantation, a facilité la mise en place d'une construction itérative en pratiquant par transformations et vérifications successives.

Bibliographie

Carroll L.F., « Vers la maîtrise du développement d'un contrôleur temps réel sûr de fonctionnement pour les robots manipulateurs », Thèse de doctorat, 1999, Institut National des Sciences Appliquées de Toulouse.

Favre J-M, Estublier J., Blay-Fornarino M., « L'ingénierie dirigée par les modèles. Au-delà du MDA », Traité IC2, série Informatique et Systèmes d'Information, Editions Lavoisier, 2006.

Gamma E., Helm R., Johnson R., et Vlissides J., « Design Patterns : Catalogue de modèles de conception réutilisables », 1999, Vuibert.

Guiochet J., « Conception Orientée objet d'un contrôleur de robot », Rapport DEA, 1999, Institut National des Sciences Appliquées de Toulouse.

Niemann S., « Executable Systems Design with UML2.0 », I-Logix™ www.ilogix.com, 2004

Object Management Group (OMG), « Model Driven Architecture Guide », version 1.0.1, juin 2003

Object Management Group (OMG), « UML Superstructure Specification », version 2.0, juillet 2004

Schraft R.D., Schmierer G., « Service Robot », AK Peters, Natick, MA (USA), 2000.

Singhoff F., Legrand J., Nana L., Marcé L., « Cheddar : a Flexible Real Time Scheduling Framework », *ACM Ada Letters journal*, 24(4):1-8, ACM Press. Also published in the proceedings of the ACM SIGADA International Conference, Atlanta, 15-18 November, 2004.

Society of Automotive Engineers (SAE) Aerospace Avionics Systems Division: « Architecture Analysis & Design Language Standards Document », version 1.0, nov. 2004.

Thomas D., « Analyse et mise en œuvre d'un contrôleur "générique" de robot manipulateur », Rapport DEA, 2004, Institut National des Sciences Appliquées de Toulouse.

Tondu B., Ippolito S., Guiochet J., Diadie A., « A Seven-degrees-of-freedom Robot-Arm Driven by Pneumatic Artificial Muscles for Humanoid Robots », *International Journal of Robotics Research*, vol. 24, n°4, MIT Press (April 2005), p. 257-274.

Génération de code pour les systèmes réactifs à partir de modèles UML2

Application sur AIBO

Xavier Blanc — Tewfik Ziadi — Cédric Besse

Laboratoire d'Informatique de Paris 6 (LIP6)
CNRS/Université Pierre et Marie Curie
{Xavier.Blanc, Tewfik.Ziadi, Cedric.Besse}@lip6.fr

RÉSUMÉ. *Dans cet article nous présentons une approche de génération complète de code pour les systèmes réactifs effectuée à partir de spécifications composées de diagrammes de classes et de diagramme de séquence UML2.0. Notre approche se découpe en deux phases : la génération d'une machine à états à partir des diagrammes de séquence puis la génération du code à partir de la machine à étvats générée. Cette approche a été mise en œuvre pour générer le code de robot AIBO (robot chien vendu par SONY). Nous présentons ici les résultats de notre travail qui permettent de mesurer les avantages et les inconvénients de UML2.0 pour la génération complète de code.*

ABSTRACT. *In this paper we present a code generation approach for reactive systems based on UML2.0 models (class diagram and sequence diagram). Our approach is composed of two steps: first, the generation of a state machine from the sequence diagrams and second, code generation from the generated state machine. We have applied our approach for AIBO code generation (robot made by Sony). We present here the results of our work that show advantages and drawbacks of the UML2.0 language in the context of full code generation.0*

MOTS-CLÉS : *IDM, Génération de code, Systèmes réactifs, UML.*

KEYWOR : *MDD, CODE GENERATION, REACTIVE SYSTEMS, UML, .*

1. Introduction

Un des avantages promis par l'ingénierie logicielle guidée par les modèles (IDM) est d'offrir un gain substantiel de productivité grâce à l'automatisation des transformations de modèles [Blanc 05]. L'IDM vise en effet à intégrer fortement les modèles dans tout le processus de production de l'application. L'objectif est d'utiliser les modèles pour générer complètement le code de l'application. Le statut des modèles change dès lors du tout au tout. Ils deviennent des éléments de production du code source des applications.

L'opération de génération de code à partir de modèles a reçu, depuis l'émergence d'UML, une attention particulière et est considérée comme la fonctionnalité fondamentale dans la majorité des outils CASE [CodeGeneration 06]. Cependant, il apparaît clairement que les générations de code offertes par les outils ne sont que partielles. Elles sont en effet principalement basées sur les diagrammes statiques (diagrammes de classes) et laissent aux développeurs le soin de compléter manuellement le code généré. Les gains en productivité sont alors très faibles car seule la structure du système est générée et non son comportement.

En soi, la génération complète de code structurel et comportemental à partir de modèles n'est pas une idée nouvelle. En particulier, cette idée a été largement développée pour les systèmes réactifs depuis [Harel 85]. Un système réactif est un système ouvert répondant constamment aux sollicitations de son environnement en produisant des actions. La complexité de développement des systèmes réactifs réside entièrement dans le code de contrôle du système (réaction aux sollicitations de l'environnement et exécution d'actions).

Ce code de contrôle peut être entièrement modélisé à l'aide de machine à états ce qui engendre des gains de productivité car le code comportemental est généré automatiquement. Cependant ces gains ne sont pas optimaux car modéliser directement le système avec des machines à états n'est pas un processus intuitif (la notion d'état n'est pas souvent naturelle à identifier dans les premières étapes de processus de développement). D'où l'intérêt d'utiliser, en plus des machines à états, d'autres formalismes comme les scénarios (ex. les diagrammes de séquence d'UML) qui sont plus intuitifs et montrent clairement les interactions du système avec son environnement [Harel 03]. Le processus de génération de code est donc guidé par les machines à états et aussi par les scénarios.

C'est dans cette optique de génération de code, guidée par les scénarios et les machines à états, que se situe notre article. Nous proposons un cadre proche de celui de Harel et al [Harel 03] pour la génération de code à partir de modèles UML2.0. L'objectif étant de mesurer la faisabilité de l'approche tout en mesurant les limites et les avantages portés par UML2.0. Cet article est organisé comme suit. La section 2 présente l'approche que nous proposons pour la génération de code à partir de modèles UML2.0. La section 3 présente l'illustration de notre approche sur la plate-forme AIBO. Dans la section 4 nous présentons les travaux connexes à notre approche puis la section 5 conclut ce travail et présente nos perspectives.

2. Génération complète de code à partir de scénarios

L'approche que nous proposons consiste à spécifier le comportement attendu du système uniquement à l'aide de diagrammes de classes et de diagrammes de séquence UML2.0. A partir de cette spécification, notre processus de génération de code se découpe en deux phases : une phase de synthèse d'une machine à états à partir des diagrammes de séquence et une phase de génération de code à partir de la machine à états générée. Cette approche ressemble fortement à celle proposée dans

[Harel 03] si ce n'est qu'elle se base entièrement sur UML2.0 (cf. figure1). Notre contribution consiste donc à proposer une approche de spécification et de génération de code guidée par les scénarios et basée sur UML2.0. Les paragraphes suivants décrivent les deux phases principales du processus de génération de code.

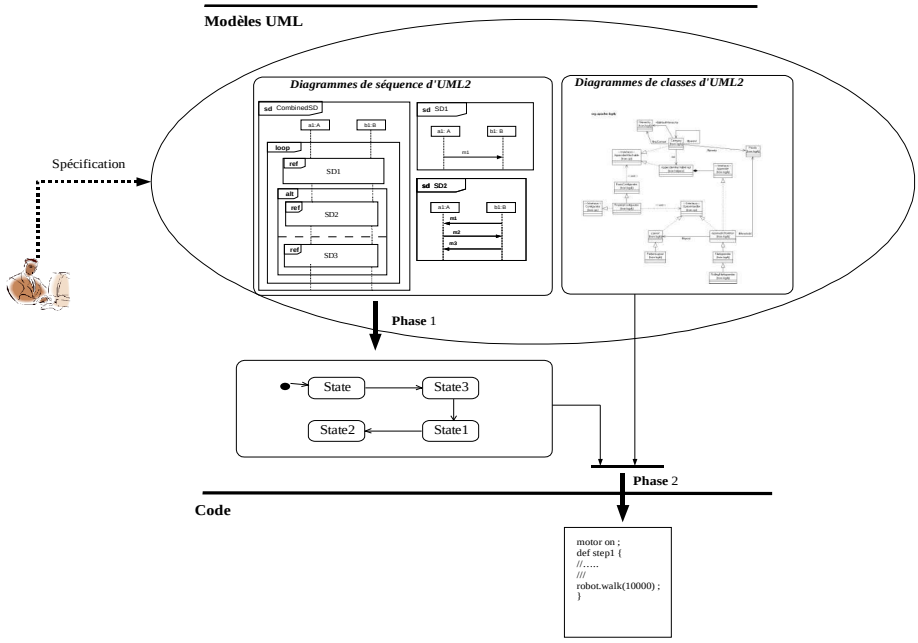


Figure 1 : Approche de génération de code pour les systèmes réactifs

La première phase de notre approche consiste à générer une machine à états représentant le comportement de la classe principale du système réactif à partir des diagrammes de séquence de la spécification. En effet, même si les diagrammes de séquence et les machines à états diffèrent dans la vue qu'ils donnent du système (un diagramme de séquence spécifie une vue inter-objets du système alors que la machine à états spécifie une vue intra-objet du système), la relation entre ces deux formalismes a attiré beaucoup d'attention et plusieurs méthodes ont été proposée pour la génération automatique de machines à états à partir de diagramme de séquence [Ziadi 04a]. Dans notre approche, nous formalisons donc la première phase comme une génération automatique de machine à états à partir des diagrammes de séquence d'UML2.0 inclus dans la spécification du système en utilisant la méthode proposée par l'un des auteurs dans [Ziadi 04a].

L'objectif de la deuxième phase est de générer le code du système à partir de la machine à états d'UML2.0 obtenue dans la première phase. Même s'il existe encore quelques différences entre les machines à états dites classiques et les machines à états UML2.0 [Crane 05], il est admis aujourd'hui que UML2.0 supporte tous les

concepts des machines à états nécessaires à la génération de code. Il semble donc possible de générer le code de systèmes réactifs à partir des machines à états UML2.0. Dans notre approche, nous proposons donc une méthode de génération de code à partir des machines à états d'UML2.0

3. Réalisation de l'approche sur AIBO

3.1. *AIBO*

3.1.1. Un système réactif

Nous avons choisi d'expérimenter la génération de code à partir de modèles UML2.0 sur une plate-forme concrète de systèmes réactifs, en l'occurrence la plate-forme AIBO de Sony. AIBO est un robot chien interactif dont le comportement peut être entièrement programmé.

D'un point de vue moteur, AIBO est composé : de quatre pattes, articulées chacune par trois moteurs, d'un cou, articulé par trois moteurs, d'une queue articulée par un moteur et de deux oreilles articulées chacune par un moteur. Grâce à ses moteurs, AIBO peut alors effectuer les actions classiques telles que se lever, s'asseoir, se coucher, marcher, etc.

AIBO interagit avec son environnement à l'aide de plusieurs capteurs : un œil représenté par une caméra située dans sa bouche, deux oreilles, différents capteurs situés sur son dos, sous ses pattes et sous sa bouche. AIBO peut donc voir et entendre et il peut toucher son environnement. AIBO peut à tout moment accéder à l'état de ses censeurs et ainsi réagir aux changements de son environnement.

AIBO est donc un système réactif à part entière qui effectue des actions (avancer, reculer, s'asseoir, …) en fonction des événements qu'il reçoit de son environnement (appel de son nom, visualisation d'une balle, …). L'intérêt de cette plate-forme est qu'elle constitue un cadre fini (par le nombre de moteurs et par le nombre de censeurs) sans pour autant être simpliste. AIBO est donc, pour nous, le cadre expérimental idéal permettant de valider notre approche.

3.1.2. Un système programmable

La programmation de comportement AIBO n'est pas une tâche triviale. Pour cela, nous utilisons URBI (Universal Real-time Behavior Interface) qui est une plate-forme de développement expérimentale pour les robots [Baillie 05]. URBI simplifie énormément la tâche de développement de programme AIBO. Il se compose d'un langage de script et d'un interpréteur. L'interpréteur s'exécute sur le robot et interprète le script transmis par le programmeur.

URBI fournit un ensemble de primitives permettant de demander à AIBO d'effectuer les actions de base (s'asseoir, se lever, se coucher, avancer, tourner à gauche, …). Par exemple, la commande URBI suivante « robot.stand() » permet de

demander à AIBO de se lever. URBI fournit aussi un ensemble de primitives permettant d'accéder aux capteurs de AIBO. Il est par exemple possible de récupérer l'image observée par AIBO. Ainsi, la commande « camera.val » permet d'obtenir le tableau d'octets représentant l'image vue par AIBO. Pour finir, grâce aux structures de contrôle proposées par URBI (boucle et gestion des événements) et grâce à la possibilité de demander l'exécution d'actions parallèles, il est alors possible de programmer des comportement interactifs. Nous présenterons plus précisément URBI dans la section relative à la génération de code à partir de machine à états.

3.1.3. Un système modélisable en UML

Afin de modéliser AIBO en UML et de pouvoir ainsi construire une génération de code basée sur les modèles, nous avons choisi de le représenter à l'aide d'une classe UML (classe AiboMove).

La figure 2 représente une partie de cette classe (pour des raisons de clarté, nous ne présentons pas l'intégralité des attributs et des opérations). Les attributs de cette classe représentent les moteurs et les capteurs de AIBO. Les opérations de cette classe représentent les actions que peut réaliser AIBO mais uniquement celles qui sont offertes sous forme de primitives URBI.

Un objet instance de cette classe représente un AIBO qui se trouve dans une position définie par les valeurs de ses moteurs et qui a une représentation de son environnement donnée par les valeurs de ses censeurs. Il est alors possible de faire effectuer des mouvements à cet AIBO en lui demandant de réaliser ces opérations.

Ainsi, afin de suivre l'approche de génération de code que nous avons présentée dans la première partie de cet article, le développeur doit utiliser cette classe pour définir les différents diagrammes de séquence représentant les comportements qu'il veut que AIBO réalise. Pour illustrer notre approche nous considérons un exemple d'un comportement attendu de AIBO. Ce comportement, illustré par les diagrammes de séquence dans les figures 3 et 4, spécifie des interactions entre l'acteur Dresseur et Apollon, un objet instance de la classe AiboMove. La figure 3 présente trois diagrammes de séquence de base : « Initialization », « Walking » et « Turning ». Le diagramme de séquence de base « Walking » par exemple montre les interactions entre l'acteur Dresseur et Apollon pour réaliser la marche. Le diagramme de séquence combiné « Sequence1 » compose les diagrammes de séquence de la figure 3 pour spécifier un comportement plus complet pour l'objet Apollon. L'opérateur `alt` permet de spécifier qu'il y a un choix entre les deux scénarios « Walking » et « Turning » en fonction de la visibilité qu'a AIBO de la balle (cf. figure 4).

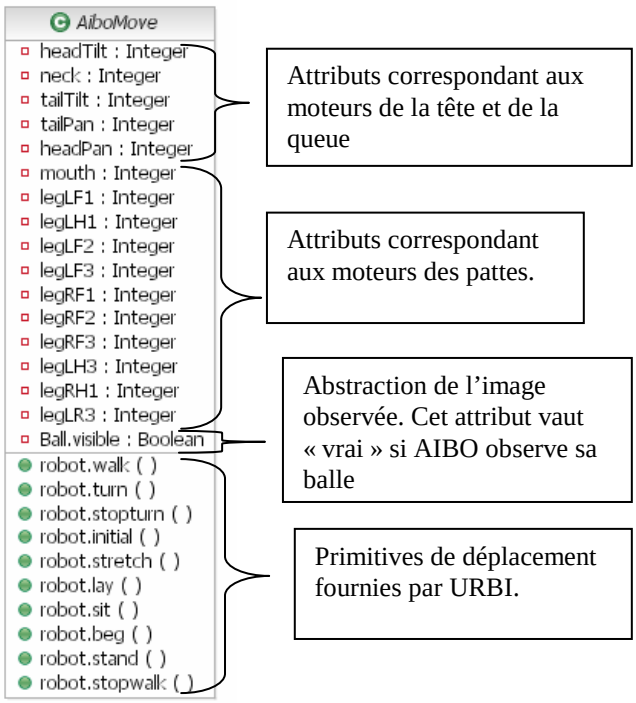


Figure 2 : la classe AIBO

Les différents diagrammes de séquence de la spécification donneront alors lieu à la création d'une machine à états. Cette machine à états représentera l'ensemble des comportements de AIBO. Il sera alors possible de générer le code URBI correspondant à cette machine à états. Les deux étapes de génération que nous détaillons dans la suite de cet article sont la génération de la machine à états à partir de plusieurs diagrammes de séquence et la génération de code à partir de la machine à états.

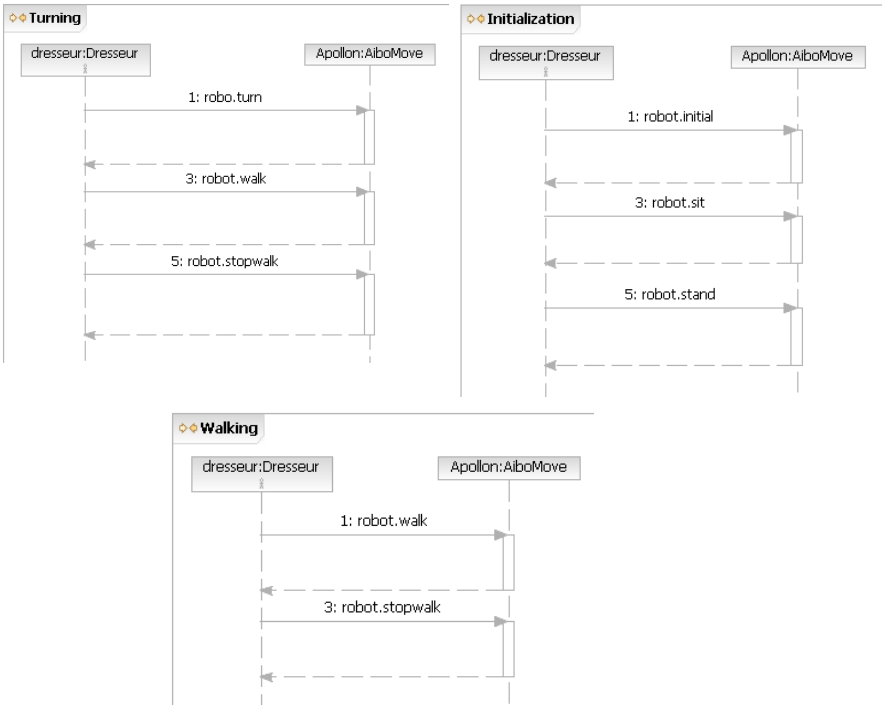


Figure 3 : Exemple de diagrammes de séquence de base pour AIBO

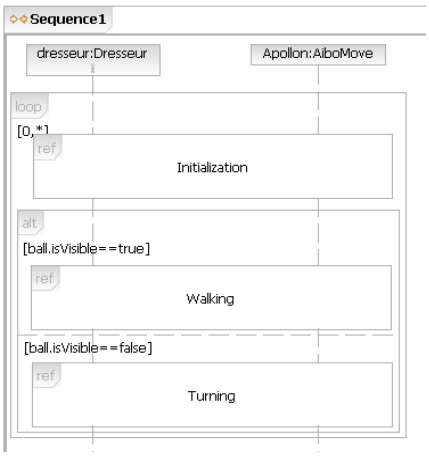


Figure 4 : Le diagramme de séquence combiné pour AIBO composant les diagrammes de séquence de base de la Figure 3.

3.2. Diagrammes de séquence vers machines à états

3.2.1. Définition de la transformation

Même si les scénarios et les machines à états diffèrent dans la vue qu'ils donnent du système (vue inter-objet vs. vue intra-objet), la relation entre les deux formalismes a attiré beaucoup d'attention avant même l'apparition d'UML. Rumbaugh a clairement établi un lien direct entre les scénarios et les machines à états dans sa méthode OMT (Object Modeling Technique) [Rumbaugh 95]. Dans le contexte d'UML, plusieurs travaux ont été proposé ces dernières années pour la génération (appelée aussi synthèse) automatique de machines à états à partir de scénarios Le lecteur trouvera une étude détaillée sur les travaux existants dans [Ziadi 04b].

L'idée principale de la synthèse de scénarios en une machine à états est de permettre la modélisation intégrale d'un système à l'aide de scénarios tout en permettant, grâce à la sémantique des machines à états, la génération de code mais aussi la vérification et la simulation du système. En d'autres termes, l'objectif par exemple est de générer le code d'un système tout en permettant à son concepteur d'exprimer ses exigences à l'aide de scénarios, ce qui est beaucoup plus intuitif pour lui.

La méthode de synthèse de machines à états que nous avons adoptée dans cet article prend comme entrée une collection de diagrammes de séquence de base (qui ne possèdent pas des références vers d'autres diagrammes de séquence) et un seul diagramme de séquence combiné (qui composent les diagrammes de séquence de base en utilisant des opérateurs d'interaction). Notre méthode, implémentée dans un outil prototype appelé PLiBS[1], se déroule en deux étapes [Ziadi 04a]. Nous générons, dans un premier temps, les machines correspondant à chaque diagramme de séquence de base. Puis, dans un second temps, nous composons ces machines à états pour obtenir la machine à états correspondant au diagramme de séquence combiné.

Pour la première étape de la génération, notre algorithme génère une machine à états pour chaque objet participant à la séquence. Pour un objet particulier, le principe de l'algorithme est de suivre la ligne de vie de l'objet et de construire les états et les transitions correspondant aux événements des messages ciblant ou partant de la ligne de vie de cet objet. .

L'algorithme commence par créer un état initial et un état d'entrée E0 et puis procède au traitement de chaque événement situé sur la ligne de vie de l'objet. Pour un évènement de réception d'un message (c'est-à-dire un événement d'appel d'opération), l'algorithme crée un nouvel état et une transition ciblant cet état.

[1] PLiBS (Product Line Behavior Synthesis) est un prototype intégré à la plate-forme Eclipse et supportant les diagrammes UML d'Omondo. PLiBS est disponible sur http://modelware.inria.fr/plibs.

L'événement déclencheur de la transition crée porte le label de l'événement en cours du traitement. La figure 5 illustre cela avec l'événement de réception de message « robot.initial ». On voit bien la création de l'état E1 et la transition entre les deux états E0 et E1.

Notons que nous avons décidé de modifier légèrement notre algorithme de synthèse proposé dans [Ziadi 04a] en ajoutant à chaque état (créé pour un événement de réception) l'activité réalisée lorsque cet état est actif (le concept de doActivity dans UML). Par exemple dans la figure 5, nous avons ajouté à l'état E1 l'activité « robot.initial ». Une autre modification de l'algorithme concerne l'ajout des états et des transitions concernant la fin des activités réalisées. Par exemple, pour l'activité « robot.initial » de l'état E1, nous avons ajouté un nouvel état E2 et une transition de E1 à E2 dont l'événement déclencheur est « endOf_robot.initial ».

La deuxième étape de notre méthode de synthèse consiste à composer les machines à états générées à partir des diagrammes de séquence de base pour obtenir une machine à états complète correspondant au comportement spécifié dans le diagramme de séquence combiné. Pour cela, nous avons proposé dans [Ziadi 04a] un cadre algébrique pour la spécification des diagrammes de séquence d'UML2.0 et de machines à états. Un diagramme de séquence combiné est spécifié algébriquement sous forme d'une expression appelée RESD (Reference Expression for Sequence Diagram). Une RESD est une expression dont les termes sont des références vers les diagrammes de séquence de base et les opérateurs sont ceux des diagrammes de séquence d'UML2.0 : `seq`, `alt`, et `loop`[2].

Pour les machines à états, nous avons formalisé trois opérateurs de composition de machines à états : $\texttt{seq}_{\texttt{st}}$(séquence) $\texttt{alt}_{\texttt{st}}$,(alternative) et $\texttt{loop}_{\texttt{st}}$ (boucle) (cf. figure 6). La composition de machine à états est spécifiée algébriquement sous forme des expressions appelées REST (Reference Expression for Statecharts). Une REST est une expression dont les termes sont des références vers des machines à états et les opérateurs sont $\texttt{seq}_{\texttt{st}}$, $\texttt{alt}_{\texttt{st}}$, et $\texttt{loop}_{\texttt{st}}$. Par manque de place, nous avons choisi de ne pas présenter dans cet article le détail de ce cadre algébrique. Néanmoins, le lecteur trouvera toute sa formalisation dans [Ziadi 04b].

En utilisant ce cadre algébrique, la composition de machines à états générées dans la première étape est basée sur une correspondance entre les opérateurs des diagrammes de séquence et ceux de machines à états. En effet, à partir de RESD correspondant au diagramme de séquence combiné de la spécification, nous construisons une REST qui permet d'obtenir une machine à états complète. Dans la section suivante nous illustrons la machine à états obtenue à partir du diagramme de séquence combinée de la figure 4.

---

[2] Notons qu'en plus de `seq`, `alt` et `loop`, UML2.0 offre d'autres opérateurs d'interaction [OMG 04]. Nous n'avons considéré dans notre méthode de synthèse que ces trois opérateurs principaux.

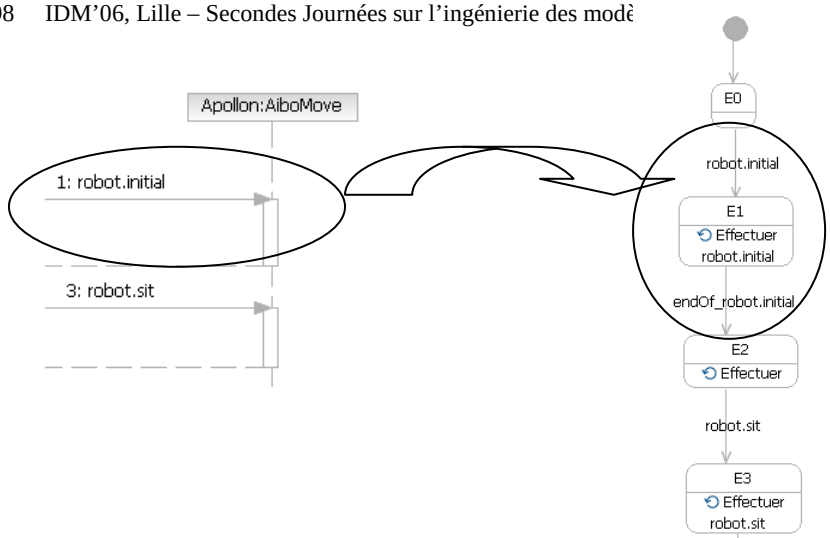


Figure 5 : Principe de la génération d'une machine à états à partir d'un diagramme de séquence de base.

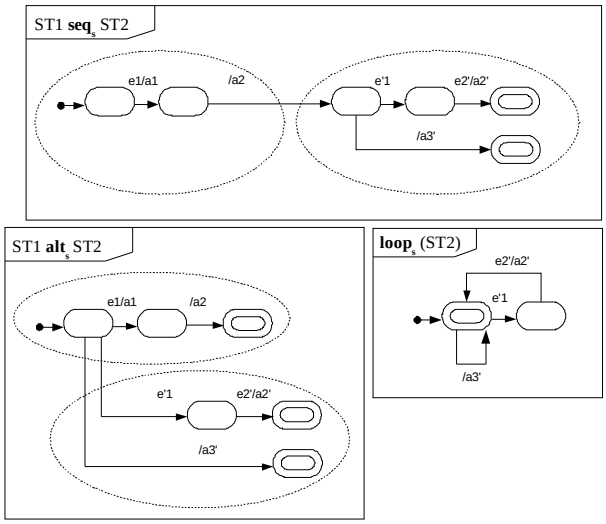


Figure 6 : Principe de la composition de machines à états.

3.2.2. Limites

La première difficulté que nous avons rencontrée concerne le traitement des messages du retour d'invocation d'opérations. En effet, notre algorithme de synthèse, défini dans [Ziadi 04a], ne considérait que les événements associés aux messages d'invocation d'opérations et non ceux associés aux retours d'invocation. Dans le contexte de AIBO, ce dernier type d'évènements est important pour générer un comportement cohérent. Nous avons alors choisi d'ajouter des événements de

type « endOf_opération » après la fin du traitement de chaque activité. Ces événements sont automatiquement envoyés lors de la fin de la doActivity [OMG 04] et permettent ainsi de bien attendre la fin de la réalisation de l'action avant de commencer quoique ce soit d'autre. Néanmoins, même si cette modification mineure de l'algorithme semble résoudre notre difficulté, elle fait ressortir le besoin de différents algorithmes de synthèse selon les objectifs souhaités (ici, le support de comportement de systèmes réactifs).

La deuxième difficulté que nous avons rencontrée concerne l'affectation des valeurs des paramètres des messages invoquant les opérations. En effet, dans le contexte des systèmes réactifs, permettre la saisie des valeurs des paramètres des opérations est absolument nécessaire. Par exemple, l'opération « robot.walk() » de AIBO permet de préciser la distance à parcourir, il est alors nécessaire de pouvoir saisir cette valeur dans les diagrammes de séquence. Le métamodèle UML2.0 propose le concept de ValueSpecification pour spécifier les valeurs des paramètres. Or, ce concept est un concept abstrait (la métaclasse est abstraite) et les concepts concrets qui en héritent ne permettent que de spécifier des valeurs littérales ou de spécifier des valeurs en utilisant un autre langage que UML. Il n'est donc pas possible de spécifier une valeur à l'aide d'une expression en utilisant uniquement UML. Dans le cadre de notre étude, nous avons donc fait le choix de ne pas supporter dans cette première version la possibilité de spécifier des valeurs pour les paramètres.

3.2.3. Réalisation de l'exemple

Pour appliquer notre méthode de synthèse sur les diagrammes de séquence des figures 3 et 4, nous utilisons en premier temps l'algorithme présenté ci-dessus pour générer des machines à états pour l'objet Apollon à partir des trois diagrammes de séquence de base (Initalization, Walking, et Turning). Dans la deuxième étape, nous composons les trois machines à états obtenues pour en construire une correspondant au diagramme de séquence combiné. La machine à états construite est illustrée dans la figure 7. Elle montre bien le comportement de l'objet Apollon à travers les différentes machines à états générées à partir des diagrammes de séquence de base. Le comportement spécifié entre l'état E0 et l'état E6 concerne la machine à états générée à partir du diagramme de séquence de base « Initialization ». De l'état E6, deux comportements alternatifs sont possibles correspondant à l'alternative spécifiée dans le diagramme de séquence combiné (cf. figure 4). Le comportement entre les états E7 et E9 est associé à la machine à états correspondant au diagramme de séquence « Walking ». Entre les états E10 et E14, nous identifions le comportement concernant le diagramme de séquence « Turning ». Notons que comme le diagramme de séquence combiné utilise l'opérateur `loop` (cf.. figure 4), la machine à états obtenue montre aussi une boucle illustrée par les transitions partants des états E9 et E14 et ciblant l'état d'entré E0.

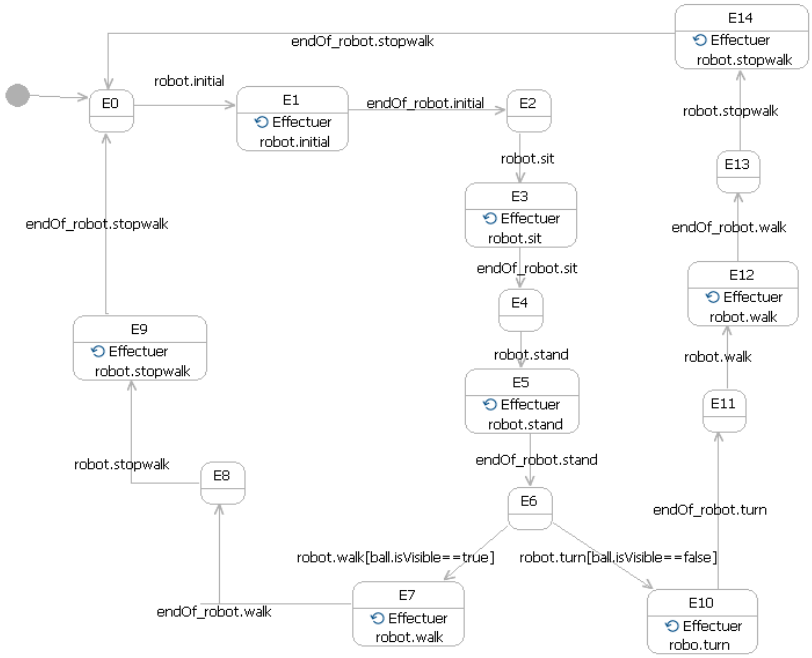


Figure 7: La machine à états de l'objet Apollon générée à partir des diagrammes de séquence des Figure 3 et 4.

3.3. Machine à états vers code

3.3.1. Principe de la transformation

La génération de code à partir de machine à états est un domaine de recherche très bien identifié mais qui reste essentiellement l'apanage des outils UML avec I-Logix en tête [I-Logix]. De plus, mise à part la description du fameux pattern state [Gamma 95] et les travaux de [Pinter 03] et de [Chauvel 05], il n'existe que peu de travaux décrivant un framework générique de génération de code à partir de machine à états. Les raisons à cela sont multiples. La première raison que nous avons identifiée vient sans doute du faite qu'il existe différentes sortes de machines à états. L-M Crane expose cela clairement dans [Crane 05]. La deuxième raison, plus pragmatique, vient de la nature très hétérogène des langages de code ciblés. En effet, même en considérant uniquement les langages orientés objet, plusieurs concepts de ces langages peuvent être utilisés pour supporter l'exécution de machine à états (exception, événements, etc.). Etant donné d'une part que nos travaux portent sur UML2.0 et d'autre part que notre langage cible n'est pas un langage objet, il nous a donc fallu proposer une approche propriétaire

Nous avons décidé d'utiliser fortement les concepts URBI de définition de fonction et de gestion d'événements. Notre idée directrice a été de définir une fonction pour chaque état de la machine à états et de faire correspondre une gestion

d'événements pour chaque transition. Plus précisément, si les événements correspondant à une transition apparaissent, alors la transition appelle la fonction correspondant à l'état qu'elle cible. Cette fonction fait appel aux comportements associés à l'état (notion de doActivity dans UML).

Associée à cette idée centrale, nous avons utilisé les mécanismes URBI permettant de préfixer les fonctions afin de contrôler leur exécution. URBI permet en effet de stopper à tout moment l'exécution d'une fonction si celle-ci a été préfixée lors de son appel. Ainsi, lorsque l'on sort d'un état, il faut stopper l'exécution de la fonction qui lui correspond. Pour finir, nous avons utilisé un mécanisme classique de variable globale permettant de connaître l'état courant (actif) de la machine à états.

La figure 8 présente les principes de génération de code que nous avons suivis. L'étape 3 donne lieu à la définition d'une fonction (nommée etape3). Cette fonction met à jour la variable globale correspondant à l'état courant de la machine à états. Puis, cette fonction appelle l'opération « robot.walk() » qui correspond à l'activité réalisée lorsque l'état est actif (doActivity dans UML2.0). La transition t3 donne lieu à l'utilisation du mécanisme de gestion d'événements (« at » dans URBI). On voit que si t3 est franchie et si l'état courant est l'état « etape2 » (qui n'apparaît pas dans le schéma), alors la fonction etape3 est appelée.

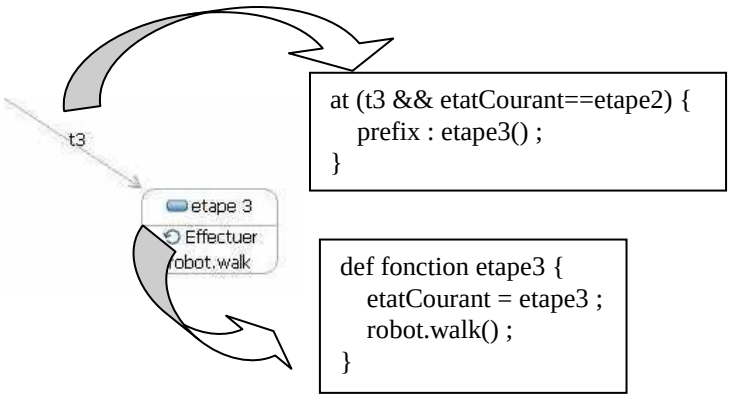


Figure 8: Principe de la génération de code URBI à partir d'une machine à états

Il est important de noter que cette génération de code URBI à partir de machine à états n'est actuellement pas complète. En autres, elle ne supporte pas encore les états composites ni la gestion des pseudo états de type « fork » ou « join ». Ce travail, même s'il est actuellement en cours de réalisation, n'est pas nécessaire pour notre propos car les machines à états construites à partir des séquences n'utilisent pas ces concepts. Notons pour finir que nous travaillons en étroite collaboration avec l'équipe de J-C. Baillie de l'ENSTA qui a créé URBI afin de construire notre génération.

3.3.2. Limites

La première difficulté que nous avons rencontrée est le manque de sémantique des machines à états UML qui est identifiée dans la littérature [Crane 05]. Pour autant, cela ne doit pas être considéré comme une lacune de UML mais plutôt comme une flexibilité du langage. Il reste alors à préciser cette sémantique lors de l'exploitation des machines à états. Pour cela, l'approche de P-A. Muller permettant de modéliser une sémantique nous paraît fortement appropriée [Muller 05].

La deuxième difficulté que nous avons rencontrée porte sur le manque de formalisation pour exprimer les gardes portées par les transitions. En effet, UML2.0 ne préconise aucun langage pour l'expression de celles-ci. Plus précisément, il définit le concept d'expression opaque permettant l'introduction dans le modèle UML d'expressions écrites avec d'autres langages. Cette difficulté majeure est bloquante. Nous l'avons contournée en utilisant le langage URBI comme langage d'expression. Notre machine à états ne peut donc être considéré réellement comme un modèle UML puisqu'elle contient des instructions URBI (uniquement des assertions). Cette problématique est, elle aussi, bien identifiée et fait l'objet d'un RFP à l'OMG. Des approches, telles que celle de S-J. Mellor qui vise à définir un langage d'action sur les modèles UML, nous semble être prometteuses [Mellor 02].

La troisième et dernière difficulté que nous avons rencontrée porte sur la gestion du temps. Le temps est, en principe, supporté par les machines à états UML2.0 mais celui-ci est clairement sous spécifié. UML ne propose en effet aucune sémantique associée au temps. Cette difficulté n'est pas majeure mais limite sensiblement les possibilités de production de l'approche. La plate-forme AIBO utilise fortement le temps. Il est par exemple possible de demander l'exécution d'un mouvement pendant une certaine durée. Notre approche ne permet donc pas de supporter cela.

3.3.3. Réalisation de l'exemple

Le code généré à partir de la machine à états de la figure 7 n'est pas très complexe. Il contient la définition de 14 fonctions (une fonction par état) et de 17 opérateurs de gestion d'événements (une par transition). Les fonctions correspondant aux états E1, E3, E5, E7, E9, E10, E12 et E14 contiennent des appels aux primitives URBI correspondant au « doActivity » des états les autres fonctions ne contiennent aucun appel. Les opérateurs de gestion d'événements correspondant aux transitions sortant de l'état E6 intègrent un test permettant de vérifier que AIBO voit sa balle. Ce code est directement interprétable par URBI et, grâce à une console, il est possible d'envoyer les événements à AIBO et observer son comportement.

4. Travaux Connexes

Comme nous l'avons mentionné dans l'introduction, l'approche que nous proposons dans cet article est similaire à celle de Harel et al [Harel 00]. La différence principale entre l'approche de Harel et la notre est la nature du langage de

scénarios utilisé. En effet, tandis que nous utilisons les diagrammes de séquence d'UML2.0 pour la spécification des scénarios, Harel utilise les Live Sequence Charts (LSC). Les LSC sont légèrement différent des diagrammes de séquence d'UML. Ils disposent des notions de « scénario universel » et « scénario existentiel » qui permettent de préciser la nature obligatoire ou non d'une suite d'actions [Harel 03]. Notons pour finir que les travaux de Harel portent moins sur la génération de code mais plus sur la simulation de modèle grâce notamment à la fameuse « play engine » qui permet de faire vivre les scénarios.

Dans le contexte d'UML, il n'existe que peu de travaux dans la littérature sur la génération de code à partir des diagrammes de séquence. Les auteurs dans [Whittle 03] présentent une approche pour la génération de code d'un système de contrôle aérien à partir de scénarios. Les auteurs appliquent la synthèse de machines à états à partir des diagrammes de séquence d'UML1.x en utilisant la méthode proposée dans [Whittle 00]. Par la suite, ils utilisent Rational Rose RealTime (prédécesseur de RSA) pour la génération de code C++ à partir de machines à états obtenues. Même si la conclusion des auteurs montre que les résultats sont satisfaisants, l'inconvénient majeur de cette approche est qu'elle ne considère que les diagrammes de séquence de base et elle ne supporte par leur composition ; donc elle n'est pas adaptée à la nouvelle version des diagrammes de séquence d'UML2.0.

5. Conclusion et Perspectives

Les travaux que nous avons réalisés montrent que l'on peut générer complètement du code pour AIBO à partir d'un ensemble de diagramme de séquence UML2.0. Il important de souligner que cette génération est possible car AIBO, ou plus précisément URBI, fournit un ensemble de primitives préprogrammées permettant d'actionner le robot. Le code généré est donc uniquement un code de contrôle permettant de lier les actions d'AIBO avec les événements qu'il reçoit de son environnement.

En souhaitant baser notre approche sur UML2.0 nous avons pu clairement identifier les lacunes de ce langage. La seule réelle ombre au tableau que nous retenons comme bloquante est le fait de ne pas disposer d'un langage d'expression formellement lié à UML. Il n'est alors pas possible par exemple d'exprimer une condition d'un « if » dans une combinaison de diagrammes de séquence ni de préciser les valeurs des paramètres lors d'un appel d'opération. Plusieurs approches sont aujourd'hui en cours d'élaboration pour faire face à cette limitation. L'approche qui nous semble être la plus prometteuse est celle qui consiste à associer formellement un langage d'actions à UML.

Nos travaux nous ont permis aussi de nous apercevoir que le langage UML2.0 n'était pas pleinement exploitable dans le domaine des systèmes réactifs. En effet, le concept d'événement envoyé par l'environnement n'est pas réellement intégré dans les diagrammes de séquence et l'approche qui consiste à le modéliser sous forme

d'un appel d'opération envoyé par un acteur externe n'est que peu satisfaisante. De plus, le temps est sous spécifié alors qu'il est largement employé dans la programmation de système réactif. Les travaux effectués sur UML-RT (RT pour Real Time) sont largement en avance sur ce domaine [Lanusse 98].

Pour conclure, même si nos travaux laissent à penser qu'une approche de génération complète de code à partir de modèle UML2.0 semble réalisable, il est encore trop tôt pour pouvoir réellement et scientifiquement se prononcer. En effet, la réalisation des quelques exemples présentés ne constitue en rien une validation de l'approche. Le travail le plus délicat qu'il reste à faire porte alors bien plus sur l'expressivité d'une telle approche que sur la possibilité à fournir un code qui tourne !

Enfin, si la méthode de génération de code que nous proposons dans cet article a été mise à l'épreuve d'un cas d'étude réaliste, rien ne permet néanmoins de certifier d'un point de vue formel la chaîne complète de génération. En d'autres termes, nous n'avons pas la garantie que les besoins définis au niveau des diagrammes de séquence UML2.0 se retrouvent intacts au niveau du code généré.

Il est alors nécessaire de trouver un moyen de montrer que les comportements définis dans les diagrammes de séquence, puis dans les machines à états UML2.0, recouvrent bien ceux du code généré. Cette préoccupation est au cœur du domaine de recherche du test des systèmes réactifs. Le test de conformité vise en effet à tester l'implémentation d'un système par rapport à sa spécification. Un ensemble de tests permettant de valider les besoins utilisateurs est généré à partir de la spécification. Ces tests sont ensuite joués sur l'implémentation, dont les réponses sont comparées aux réponses attendues définies dans les tests. De nombreuses méthodes de génération de tests à partir de machines à états ont vu le jour. D.Lee et M.Yannakakis sont d'ailleurs les auteurs d'un survey [Lee 96] sur le test à partir de machines à états finis (Finite State Machine) qui est le plus largement cité.

Il nous semble possible, en jouant sur les points de variation sémantique propres aux machines à états UML2.0 [Chauvel 05], d'identifier leur sémantique à celle des FSM, et ainsi d'adapter des techniques de génération de test à partir des FSM pour la génération de test à partir des machines à états UML2.0. Cela nous fournirait un moyen de valider non le processus de génération de code mais le code généré par rapport à sa spécification exprimée sous la forme d'une machine à états UML2.0.

5. Bibliographie

[Baillie 05] J-C. Baillie, « URBI: Toward a Universal Robotic Low-Level Programming Language », in Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS'05, Edmonton Canada.

[Blanc 05] X. Blanc, « MDA en action », Eyrolles 2005, ISBN-2-212-11539-3

[Chauvel 05] F. Chauvel, J-M. Jézéquel, , « Code Generation from UML Models with Semantic Variation Points », in Proceedings of the IEEE/ACM 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2005.

[CodeGeneration 06] Site Web, 2006, http://www.codegeneration.net/tiki-index.php

[Gamma 95] E. Gamma & al. « Design Patterns », Addison-Wesley 1995, ISBN 0-201-63361-2

[Harel 85] D. Harel, A. Pnueli, « On the development of reactive systems». Logics and Models of Concurrent Systems (1985), K. R. Apt, Ed., vol. F13 of NATO ASI Series, Springer-Verlag, pp. 477--498.

[Harel 03] D. Harel, R. Marelly, « Come, Let's Play », Springer 2003, ISBN 3-540-00787-3

[Crane 05] Michelle L. Crane, J. Dingel, « UML Vs. Classical Vs. Rhapsody Statecharts : Not All Models Are Created Equal », in Proceedings of the IEEE/ACM 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2005.

[I-Logix] I-Logix, Inc, Products Web Page. http://www.ilogix.com/

[Lanusse 98] A. Lanusse, S. Gérard, F. Terrier « Real-Time Modeling with UML: The ACCORD Approach », UML'98.

[Lee 96] D. Lee, M. Yannakakis, " Principles and Methods of Testing Finite State Machines – A Survey". In Proc. Of the IEEE, 84(8):1090-1123, August 1996.

[Mellor 02] S-J. Mellor, « Executable UML: A Foundation for Model Driven Architecture », Addison-Wesley Professional; 1st edition (May 15, 2002), ISBN- 0201748045

[OMG 04] Object Management Group OMG. « Uinified Modeling Language specification version 2.0: Superstructure ». Technical Report pct/04-08-02, OMG, 2004

[Pinter 03] G. Pinter, I. Majzik, « Program code generation based on UML statechart models », Periodica Polytechnica SER. EL. ENG. Vol, 47, N°3-4, pp.187-204, 2003.

[Rumbaugh 95] Rumbaugh et al. « Modélisation et conception orientées objet ». Masson, 1995.

[Whittle 03] J. Whittle, J. Saboo, and R. Kwan. « From scenarios to code : An air trafic control case study ». In Proceeding of International Conference on Software Engineering (ICSE 2003), 2003.

[Whittle 00] J. Whittle and J. Schumann. « Generating statechart designs from scenarios ». In Proceeding of International Conference on Software Engineering (ICSE 2000), 2000.

[Ziadi 04a] T. Ziadi, L. L. Hélouët, and J-M. Jézéquel. « Revisiting statechart synthesis with an algebraic approach ». In International Conference on Software Engineering, ICSE'04, Edinburgh, Scotland, United Kingdom, May 2004.

[Ziadi 04b] T.Ziadi., « Manipulation de lignes de produits en UML », Thèse de doctorat, université de Renne1, 2004.

Actes de l'atelier de travail
Motifs de méta-modélisation

Actes de l'atelier de travail
Motifs de méta-modélisation

Atelier de travail *Motifs de métamodélisation*

Raphaël Marvie[1] — **Jérôme Delatour**[2] — **Guillaume Savaton**[2]

[1] *Laboratoire d'Informatique Fondamentale de Lille*
UMR CNRS 8022
Université des Sciences et Techniques de Lille
Bâtiment M3 – UFR d'IEEA
F-59655 Villeneuve d'Ascq
marvie@lifl.fr

[2] *ESEO, Equipe de recherche TRAME*
4, rue Merlet de la Boulaye
BP 30926
F-49009 Angers cedex 01
{jerome.delatour,guillaume.savaton}@eseo.fr

RÉSUMÉ. *Le développement croissant de la définition et l'utilisation de métamodèles rend nécessaires l'identification et la diffusion de bonnes pratiques et de motifs. Cet atelier de travail a pour objectif d'initier un travail de collecte et de réflexion afin de produire un catalogue de ces motifs et bonnes pratiques.*

ABSTRACT. *The increasing development in the definition and use of metamodels arise the need for the identification and promotion of metamodeling best practices and patterns. This workshop aims at initiating their identification and collecting in order to set up a catalog.*

MOTS-CLÉS : *Motifs de métamodélisation, bonnes pratiques.*

KEYWORDS: *Metamodeling Patterns, Best Practices.*

1. Introduction

L'utilisation de métamodèles représente, avec les transformations de modèles, une des fondations de l'Ingénierie Dirigée par les Modèles (IDM). La définition de ces métamodèles est une activité fondamentale permettant entre autres choses la spécification des concepts d'un domaine d'application, la spécification d'une technologie d'exécution, la spécification d'un processus de développement ou d'une activité quelle qu'elle soit. Le nombre de métamodèles est en croissance exponentielle (publications scientifiques, projets industriels ou au sein des organismes de standardisation comme

l'*Object Management Group*). Toutefois, il y a actuellement peu de valorisation et de diffusion de bonnes pratiques pour la définition de métamodèles (métamodélisation).

L'identification de bonnes pratiques en conception de logiciels orientés objet s'est notamment matérialisée par la cartographie d'un certain nombre de patrons de conception (*Design Patterns*). Ces derniers représentent aujourd'hui un très bon vecteur de partage de bonnes pratiques, qu'elles soient générales (par exemple les patrons du GOF [GAM 95]) ou spécifiques à un domaine particulier (par exemple les patrons du POSA [BUS 96, SCH 00]).

2. Motivations

L'activité de métamodélisation n'est pas aujourd'hui aussi répandue que l'activité de modélisation. En effet, un métamodèle, une fois défini, est destiné à être utilisé par plusieurs personnes : une fois un métamodèle du domaine de la santé défini, il est utilisé pour modéliser les différents services médicaux et on ne re-définit pas un nouveau métamodèle pour chaque service. Il est parfois nécessaire de devoir faire une variante d'un métamodèle pour l'adapter à une utilisation particulière, mais la majorité des concepteurs de logiciels font de la modélisation et non de la métamodélisation.

Cependant, l'identification de bonnes pratiques et la formalisation de motifs de métamodélisation [MAR 04] représentent un objectif important de la communauté dans le sens où ils devraient permettre d'accélérer l'écriture de métamodèles, de faciliter leur réutilisation, et enfin leur enseignement. Tout comme pour les patrons de conception, les règles et bonnes pratiques de métamodélisation peuvent être générales ou spécifiques à un domaine : les règles de définition d'un métamodèle pour une technologie d'exécution ne seront pas identiques aux règles de définition d'un secteur d'activité (comme la santé, le commerce électronique ou la finance).

Un métamodèle est un modèle dont l'utilisation est particulière dans le sens où il sert à construire des modèles et non des systèmes (bien que l'existence d'une différence entre ces deux activités soit discutable) [FAV 06]. En cela, quels sont les similitudes, les différences et les liens de parenté entre les patrons de conception et les motifs de métamodélisation ? Il apparaît évident qu'il existe des motifs récurrents dans la définition de métamodèles, qu'il y a des règles à suivre et des pièges à éviter. Tous ces éléments sont aujourd'hui encore mal identifiés ou appliqués de manière inconsciente par les "bâtisseurs de métamodèles".

Nous pensons donc qu'il est important de prendre le temps d'étudier dès maintenant non seulement des métamodèles existants mais aussi les démarches utilisées pour leur construction afin de commencer à identifier des motifs de métamodélisation et à en faire un (ou plusieurs) catalogues pour partager cette connaissance latente. Cette activité devrait nous permettre de mieux cerner l'activité de métamodélisation et de faciliter son enseignement dans les cycles *Master* de nos universités et écoles.

3. Vers un catalogue de motifs

Toutes ces ambitions ont motivé la tenue de ce premier atelier de travail sur les motifs de métamodélisation. Parmi les soumissions reçues, cinq ont été retenues. Certaines présentent des motifs génériques pouvant être utilisés dans la définition de métamodèles pour des langages existants ou de nouveaux langages, d'autres sont plus spécifiques à des domaines particuliers.

Le motif *Elément nommé*, placé en tête de ce recueil, a été proposé comme exemple en accompagnement de l'appel à soumission. Il présente une organisation typique de métamodèle dans laquelle des éléments reçoivent un nom ou un identifiant rattaché à un espace de nommage. Outre le fait de constituer un premier exemple de motif permettant d'expliciter le type de contributions attendues, le rôle de cet article était de fixer le plan souhaité pour les soumissions.

On reconnaîtra dans ce plan l'organisation typique des catalogues de patrons de conception objet. On notera cependant que les sections "avantages" et "inconvénients" ont été volontairement omises. Du fait de la jeunesse de ce domaine, une analyse critique de ces motifs nous a semblé difficile en l'absence d'un catalogue plus étoffé et d'un véritable débat concernant les bonnes pratiques de métamodélisation. En organisant cet atelier, nous avons précisément souhaité mettre en place les conditions de ce débat.

Voici un aperçu des contributions retenues :

– Le motif *Relation, relation dirigée, association* traite de la représentation des relations entre éléments de modèles. La notion d'association, présente par exemple dans UML, peut en effet prêter à différents choix de représentation.

– La majorité des langages de programmation et de modélisation fournissent un moyen de réutiliser du "code" (classes, sous-programmes, composants). Le motif *Élément instanciable* s'intéresse à la manière dont un métamodèle peut être organisé pour permettre la réutilisation de fragments de modèles.

– Le motif *Interfaçage entre contextes* généralise les notions d'espaces de nommage, de paquetages, etc. présentes dans de nombreux langages, et traite de la question de la visibilité des éléments d'un modèle.

– Le motif *Plate-forme d'exécution* identifie les éléments nécessaires à la modélisation et à la composition des ressources permettant l'exécution d'une application.

– Enfin, l'article *Des métamodèles au banc d'essai des patrons de conception* propose un approche originale consistant à identifier dans les métamodèles les patrons typiques de la conception logicielle orientée objet.

Nous espérons que cette première manifestation va initier une activité de plus longue haleine autant sur l'identification de nouveaux motifs, sur l'utilisation de ces motifs (par exemple au travers de la définition de règles de compositions) que sur l'enseignement de la métamodélisation.

4. Bibliographie

[BUS 96] BUSCHMANN F., MEUNIER R., ROHNERT H., SOMMERLAD P., STAL M., *Pattern-Oriented Software Architecture, Volume 1 : A System of Patterns*, John Wiley and Sons, 1ère édition, 1996, ISBN : 0471958697.

[FAV 06] FAVRE J.-M., ESTUBLIER J., BLAY M., Eds., *L'Ingénierie Dirigée par les Modèles : au-délà du MDA*, Hermes-Lavoisier, février 2006, ISBN : 2746212137.

[GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., BOOCH G., *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Westley Professional Computing, USA, 1995.

[MAR 04] MARVIE R., « Vers des patrons de métamodèlisation, Structuration de métamodèles par séparation des préoccupations », *Technique et Science Informatique (TSI)*, vol. 23, n° 10, 2004, p. 1355-1382.

[SCH 00] SCHMIDT D., STAL M., ROHNERT H., BUSCHMANN F., *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*, John Wiley and Sons, 1ère édition, 2000, ISBN : 0471606952.

Motif pour la métamodélisation

Élément nommé

Guillaume Savaton, Jérôme Delatour

ESEO, équipe TRAME (TRAnsformations de Modèles pour l'Embarqué)
4 rue Merlet de la Boulaye – 49009 Angers
{guillaume.savaton,jerome.delatour}@eseo.fr

RÉSUMÉ. *Le motif Elément Nommé s'applique à la création de métamodèles dans lesquels des éléments sont identifiés et désignés par un nom unique à l'intérieur d'un espace de nommage.*

MOTS-CLÉS : *métamodèles, langage dédié, patron, motif, nommage d'élément*

KEYWORDS: *metamodel, domain specific language, model, pattern, named element*

1. Motivation

La majorité des langages de modélisation ou de programmation permettent de définir et de manipuler des *objets* auxquels on associe un *nom* ou un *identifiant* : dans les langages de programmation, les types, les variables, les sous-programmes, sont nommés de manière à pouvoir être explicitement référencés dans les parties du programme où ils sont utilisés. On distingue ainsi dans un programme la *déclaration* ou *définition* d'un élément, et *l'utilisation* de ce même élément. Suivant le langage et le contexte d'utilisation, *l'identifiant* pourra être : un nom lisible et porteur de sens pour un être humain ; un URI ; une clé numérique ou alphanumérique déterminée automatiquement ; etc.

Un élément nommé est défini dans un certain *contexte* : une variable locale est définie dans une fonction, un attribut est défini dans une classe, une classe est définie dans un paquetage, etc. Dans les exemples cités, le *contexte* de définition d'un élément nommé joue le rôle d'un *espace de nommage* : il conditionne généralement la *portée* des noms et les règles à appliquer en cas d'homonymie. La notion d'espace de nommage est associée, selon les langages, à des règles sémantiques et des mécanismes de résolution (import, noms qualifiés) permettant de retrouver l'élément désigné par un nom. Ces règles et mécanismes ne seront pas traités ici.

La notion de *nom* ou *d'identifiant* relève de la syntaxe concrète des langages. Dans un modèle MOF, on cherche au contraire à faire abstraction des mécanismes qui permettent d'associer des éléments entre eux : les systèmes de dépôts de modèles gèrent ces associations de manière transparente et fournissent des API permettant de naviguer dans un modèle sans se soucier de savoir comment les éléments sont identifiés. La notion d'identifiant réapparaît dans les formes sérialisées XMI des modèles sous la forme d'attributs *id* et *idref* [OMG-XMI].

2. Indications d'utilisation

On utilisera le motif Élément *nommé* à chaque fois que l'on voudra établir un métamodèle d'un langage de programmation ou de modélisation dans lequel la notion de *déclaration* ou *définition* d'un élément est séparée de la notion *d'utilisation* de ce même élément. Dans la syntaxe concrète d'un tel langage, un élément est typiquement associé à un nom ou identifiant au moment de sa définition ; c'est ce même nom ou identifiant qui sert à désigner l'élément à chaque fois qu'il est utilisé.

Ce motif se prête particulièrement à la métamodélisation des langages textuels, mais n'est pas limité à ce domaine. Les notions d'élément nommé et d'espace de nommage se retrouve notamment dans UML [OMG-UML] et SDL [FISCH-SDL].

3 Structure

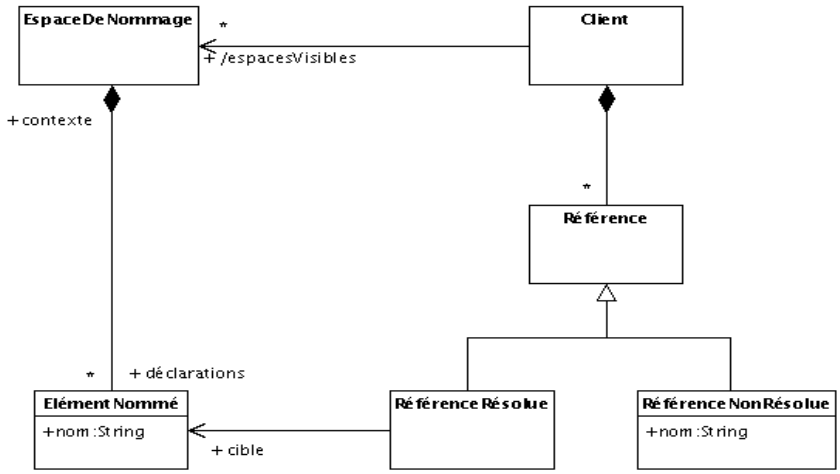


Figure 1. *Structure du métamodèle « Éléments nommés »*

4. Constituants

4.1. *ElémentNommé*

Un élément nommé représente tout élément auquel on souhaite faire référence par son nom ou son identifiant. Ce dernier est ici représenté par l'attribut *nom*.

4.2. *EspaceDeNommage*

Un espace de nommage est le *lieu* auquel sont rattachés un ensemble d'éléments nommés. Les espaces de nommage sont fréquemment utilisés pour résoudre les problèmes d'homonymie. A ce titre, il est fréquent qu'un espace de nommage soit lui-même un élément nommé.

4.3. *Client*

Le client manipule des éléments nommés par l'intermédiaire de *références*. Les références doivent désigner des éléments nommés qui appartiennent à des espaces de nommage visibles du client. Les règles qui déterminent la visibilité d'un élément sont propres à chaque langage : l'extrémité d'association *espacesVisibles* est ainsi marquée comme dérivée.

4.4. *Référence*

Une référence désigne un élément nommé, soit en l'appelant par son nom (référence *non résolue*), soit en pointant directement vers l'élément cible (référence *résolue*). L'opération de résolution des noms est une transformation de modèle qui consiste, pour chaque référence *non résolue*, à rechercher l'élément qu'elle désigne et à créer une référence *résolue*.

4. Exemple

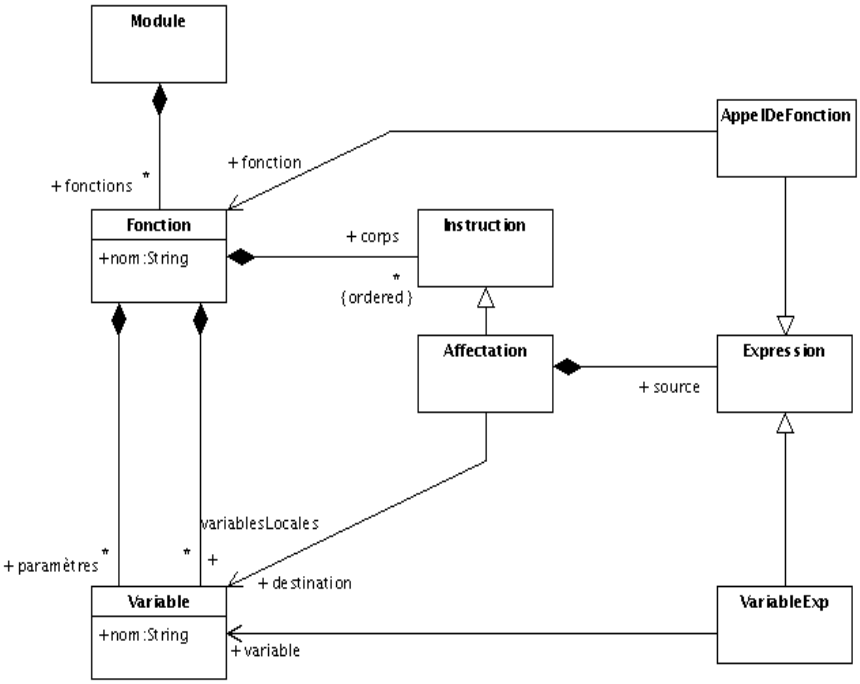
L'exemple ci-dessous est un extrait du métamodèle d'un langage procédural tel que le langage C. L'analyse syntaxique d'un programme peut être conçue en deux temps : un premier temps au cours duquel les déclarations et définitions de variables et de fonctions sont analysées ; et un second temps visant à rattacher les expressions qui utilisent des variables et des fonctions aux déclarations correspondantes. Cet extrait de métamodèle se situe après cette seconde phase.

Le motif se retrouve à travers les classes suivantes :

– Les classes *Fonction* et *Variable* jouent le rôle de la classe *ElémentNommé* du motif : les *fonctions* et les *variables* sont identifiées par un *nom* qui sert à les désigner dans le reste du programme.

– Les classes *Module* et *Fonction* jouent le rôle de la classe *EspaceDeNommage* du motif : les fonctions sont définies dans le contexte d'un module ; les variables locales et les paramètres sont définis dans le contexte d'une fonction.

– Les classes *Affectation*, *VariableExp* et *AppelDeFonction* jouent le rôle de la classe *Référence* du motif : une instruction *d'affectation* désigne la variable dans laquelle doit être rangé un résultat ; dans une *expression*, les variables utilisées et les



fonctions appelées sont désignées par leur nom.

Figure 2. *Fragment du métamodèle d'un langage procédural*

5. Bibliographie

[OMG-UML] OMG, « Unified Modeling Language: Superstructure », v 2.0, Août 2005

[OMG-XMI] OMG, « MOF 2.0/XMI Mapping Specification », v2.1, Septembre 2005

[FISCH-SDL] Joachim Fischer, Michael Piefel, and Markus Scheidgen, « A Metamodel for SDL-2000 in the Context of Metamodelling » ULF – Proc. Fourth SDL and MSC Workshop (SAM'04), June 2004

Motifs pour la métamodélisation

Relation, relation dirigée, association

Cédric Dumoulin, Arnaud Cuccuru, Antoine Honoré

LIFL, Université de Lille

cedric.dumoulin@lifl.fr, arnaud.cuccuru@lifl.fr, antoine.honore@lifl.fr

RÉSUMÉ. La conception de métamodèle introduit un certain nombre de problèmes récurrents : à chaque développement de nouveaux métamodèles des questions identiques se posent. Les solutions retenues sont très similaires, quand elles ne sont pas un simple copié/collé. Cet article présente le motif « relation » et ses déclinaisons « relation dirigée » et « association » que nous avons identifié lors de la conception de métamodèles pour la modélisation de systèmes embarqués, et pour la réalisation de moteurs de transformation. Les motifs de relations permettent de modéliser des relations entre des concepts, et d'associer de l'information à ses relations.

MOTS-CLÉS : métamodèles, patron, motif, pattern, relation, relation dirigée, association,.

KEYWORDS : metamodels, pattern, relationship, directed relation, association.

1. Motivation

Dans un modèle conforme à un métamodèle, il est courant d'avoir à matérialiser une *relation* entre des concepts, et de vouloir associer de l'information à cette *relation* comme le nom de la relation, le nom et l'arité de l'extrémité, les données propre à un bus de données (taille, bande passante, latence, …) quand la relation matérialise un tel bus, …. Dans un outil graphique de modélisation (de niveau M1), cette relation est matérialisée par un trait entre les concepts mis en relation.

On distingue plusieurs types de relations :

- les relations binaires – pour relier deux concepts.
- les relations n-aires – pour relier deux ou plusieurs concepts.
- les relations dirigées – pour relier deux concepts, l'un sert de source et l'autre de cible. Graphiquement, on représente généralement cette relation par trait avec une flèche à une des extrémités. La flèche matérialise le sens de la relation, elle pointe vers la cible.

- les associations – relations dans lesquelles ont veut être capable de mettre de l'information sur les extrémités de la relation. Par exemple, on veut pouvoir nommer l'extrémité d'une relation.

1.1. Indication d'utilisation

Les motifs présentés permettent de définir des relations dans un métamodèles (au niveau M2), et de les utiliser dans des modèles (de niveau M1).

On utilisera, au niveau M2, le motif « *relation* » chaque fois que l'on veut autoriser, au niveau M1, la mise en relations de deux ou plusieurs éléments du modèle, et que l'on veut associer de l'information à cette relation.

On utilisera, au niveau M2, le motif « *relation dirigée* » chaque fois que l'on autoriser, au niveau M1, la mise en relation dirigée de deux ou plusieurs concepts du modèle, et que l'on veut associer de l'information à cette relation. Certain concept servent de source à la relation, et les autres servent de cible.

On utilisera, au niveau M2, le motif « *association* » chaque fois que l'on veut mettre en relation, au niveau M1, deux ou plusieurs concepts du modèle, et que l'on veut associer de l'information à cette relation ainsi qu'aux extrémités de l'association. Ce motif s'appelle « association » car il est utilisé dans la modélisation de l'association UML.

Les relations peuvent être binaire ou n-aire.

2. Structures

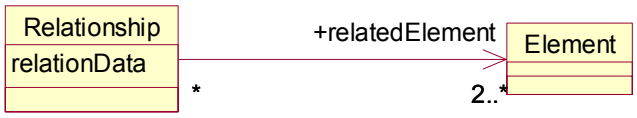


Figure 1. *Structure du métamodèle « relation »*

Figure 2. *Structure du métamodèle « relation dirigée »*

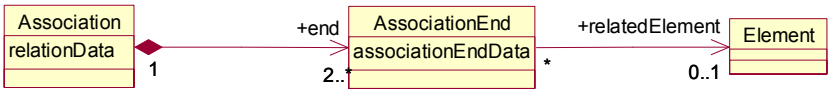


Figure 3. *Structure du métamodèle « association »*

3. Constituants

3.1. Relationship

Défini le concept de relation. Ce concept contient les informations liées à la relation. La relation connaît les éléments en relation, mais ces derniers ne connaissent pas obligatoirement la relation.

Attribut :

- relationData – donnée propre à la relation. D'autres données peuvent être associés à la relation.

3.2. Element

Un élément participant à une relation. Représente un élément quelconque du métamodèle.

3.3. DirectedRelationship

Défini le concept de *relation dirigée*. Ce concept contient les informations liées à la relation. La relation connaît les éléments reliés, mais les éléments reliés ne connaissent pas nécessairement la relation.

Attribut :

- relationData - donnée propre à la relation. D'autres données peuvent être associés à la relation.

Relations :

- source – Les éléments participant en temps que sources de la relation.
- target - Les éléments participant en temps que cible de la relation.

3.4. Association

Défini le concept *d'association*. Une association est une relation entre des éléments. De l'information arbitraire est associée à la relation, ainsi qu'à chaque extrémité de la relation. L'association connaît indirectement (par les AssociationEnd) les éléments reliés, mais les éléments reliés ne connaissent pas nécessairement les associations dont ils font partis

Attribut :

- relationData - donnée propre à la relation. D'autres données peuvent être associés à la relation.

Relation :

- end – référence les concepts contenant l'information associée à chaque extrémité de la relation. Le concept contient aussi la relation vers l'élément relié.

3.5. AssociationEnd

Classe utilisée pour contenir l'information associée à une extrémité d'une association. Cette classe référence l'élément relié par l'association.

Attribut :

- associationEndData – l'information associée à l'extrémité de l'association. D'autres données peuvent être ajoutées à l'aide d'autres attributs.

Relation :

- relatedElement – l'élément relié par cette extrémité de l'association.

4. Exemples

Les motifs relation, relation dirigée et association sont très utilisés dans la définition des métamodèles. Le premier exemple montre leurs utilisations dans UML 2. Le second exemple montre une autre utilisation dans un métamodèle pour les systèmes embarqués.

4.1. Relation et relation dirigée UML 2

Le métamodèle UML 2 Superstructure définie plusieurs déclinaisons des motifs de relations : *association UML2, ElementImport, PackageImport, PackageMerge, Dependency*, *Generalization*… Ces déclinaisons sont généralement définies par héritage d'une des classes de base, *Relationship* ou *DirectedRelationship*.

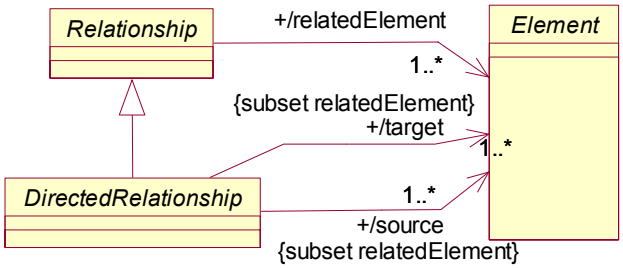


Figure 4. *Métamodèle des classes de bases pour les relations UML2*

La classe Relationship sert de classe de base pour les relations simple, tandis que la classe DirectedRelationShip sert de base pour les relations dirigées. Chaque classe du métamodèle UML 2 joue le rôle de la classe du même nom dans les motifs «relation et relation dirigée ». En UML 2, les extrémités des associations du coté de *Element* sont dérivées (indiqué par « / »). La propriété concrète correspondante sera spécifiée dans la sous-classe.

Une classe concrète est la classe UML 2 *Generalization* (Figure 5). La source et la cible sont spécifiées comme étant « *general* » et « *specific* ». La figure de droite montre un exemple d'utilisation de la généralisation.

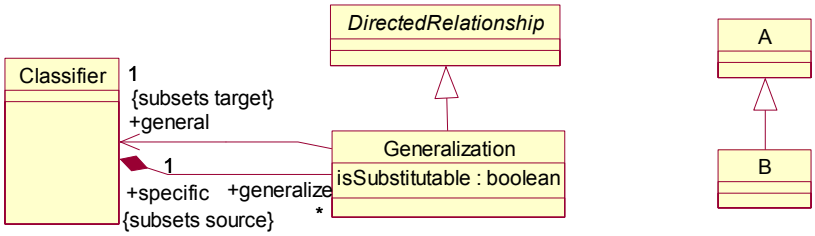


Figure 5. *Métamodèle de la Generalisation UML 2 et exemple d'utilisation*

4.2. Connecteurs UML 2

La définition des connecteurs UML 2 (Figure 6) est conforme au motif « association ». La classe *Connector* joue le rôle de *Association*, *ConnectorEnd* le rôle de *AssociationEnd*, et *ConnectableElement* le rôle de *Element*. L'information est associée au connecteur et aux extrémités à l'aide des classes *Property* et *Association*.

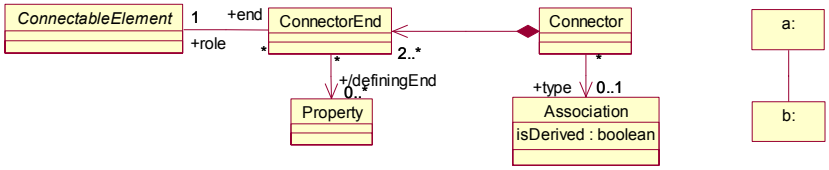


Figure 6. *Métamodèle des Connectors UML 2 et exemple d'utilisation*

4.3. Relation de dépendance « Reshape »

L'exemple suivant montre un extrait d'un métamodèle pour les systèmes embarqués. Il représente une dépendance « *Reshape* », utilisé pour matérialiser une dépendance de données entre deux composant. De l'information est associée à cette dépendance (*elementType*) ainsi que aux extrémités (*paving, fitting et origin*).

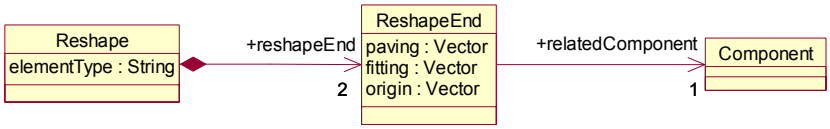


Figure 7. *Exemple « d'association » : métamodèle de la relation Reshape.*

La classe *Reshape* joue le rôle de la classe *Association*, *ReshapeEnd* le rôle de *AssociationEnd* et *Component* est l'élément mis en relation.

La figure suivante montre un exemple d'utilisation de la Relation *Reshape*. La relation est matérialisée par le trait reliant les deux composants. Les données des extrémités et de la relation sont affichées sur le diagramme.

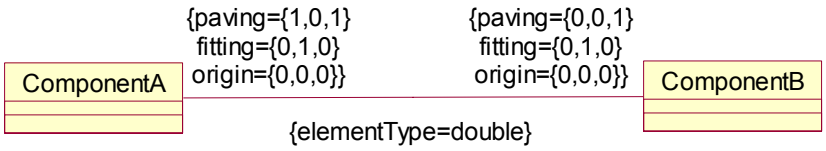


Figure 8. *Exemple d'utilisation de la relation « Reshape »*

5. Bibliographie

[OMG-UML] OMG, « Unified Modeling Language: Superstructure », v 2.1, ptc/06-01-02

[OMG-XMI] OMG, « MOF 2.0/XMI Mapping Specification », v2.1, Septembre 2005

Motif pour la métamodélisation

Élément instanciable

Guillaume Savaton

ESEO, équipe TRAME (TRAnsformation de Modèles pour l'Embarqué)
4 rue Merlet de la Boulaye – BP 30926 – 49009 Angers cedex 01
guillaume.savaton@eseo.fr

RÉSUMÉ. *Employé dans une définition de syntaxe abstraite pour un langage existant ou spécifique à un domaine, le motif "Élément instanciable" permet de séparer la description des éléments instanciables (types, classes, composants, sous-programmes) et les instanciations de ces éléments (données, objets, instances de composants, appels de sous-programmes).*

ABSTRACT. *Used in the context of an abstract syntax definition for an existing language or a domain-specific language, the model pattern "Instantiable Element" allows to separate the description of instantiable elements (types, classes, components, subprograms) and the instantiation of these elements (data, objects, component instances, subprogram calls).*

MOTS-CLÉS : *langage, syntaxe abstraite, déclaration, type, instanciation, instance.*

KEYWORDS: *language, abstract syntax, declaration, type, instantiation, instance.*

1. Motivation

La recherche de l'efficacité dans les activités de programmation et de modélisation passe en grande partie par la possibilité de capitaliser et de réutiliser des fragments de programmes ou de modèles. De nombreux langages de programmation ou de modélisation permettent de séparer, et de regrouper en bibliothèques, des fragments réutilisables qu'il sera possible d'invoquer, d'instancier à volonté dans des programmes ou des modèles spécifiques à des problèmes donnés.

La sémantique associée à la réutilisation dépend fortement du domaine métier : les langages de programmation procéduraux permettent de séparer la définition d'un sous-programme, et les appels à ce même sous-programme ; la réutilisation consiste alors à insérer dans le flot d'exécution d'un programme un fragment de comportement défini ailleurs. Les langages à objets fournissent les notions de classe et d'instance : la réutilisation consiste à créer et à faire interagir des objets conformes à une spécification de structure de données et de comportement. Les langages de description de matériel comme VHDL ou Verilog permettent de décrire des

composants matériels réutilisables qu'il sera possible d'instancier et d'interconnecter pour construire des circuits plus complexes : la réutilisation porte alors sur le comportement et la structure.

Indépendamment de la syntaxe concrète et de la sémantique de ces langages, un motif commun peut être identifié dans la démarche de réutilisation :

- tout d'abord, le concepteur définit (ou déclare) un élément réutilisable et les propriétés qui pourront être personnalisées (paramètres d'un sous-programme, attributs d'une classe, ...) ;
- dans un second temps, cet élément réutilisable est instancié, ou invoqué, et des valeurs effectives sont affectées à ses propriétés personnalisables.

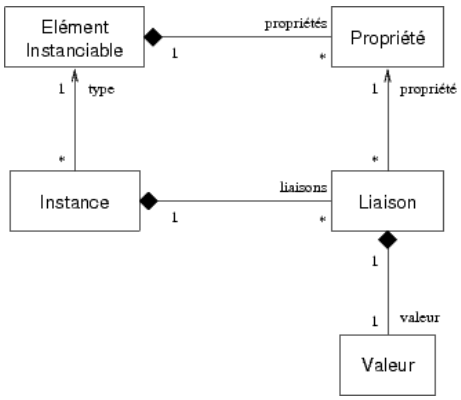
Le motif de métamodélisation que nous présentons ici propose une manière typique d'organiser la syntaxe abstraite d'un langage supportant la réutilisation.

2. Indications d'utilisation

Ce motif peut être utilisé pour établir un métamodèle d'un langage existant ou d'un DSL supportant la réutilisation de fragments de modèles.

Ce motif ne préjuge pas de la sémantique associée à la réutilisation dans le langage considéré. Il ne retient que les éléments communs identifiés dans les syntaxes abstraites de nombreux langages de programmation et de modélisation. A ce titre, il est particulièrement adapté à la définition de métamodèles dans lesquels figurent les notions de sous-programmes, types de données abstraits, composants, mais cette liste est loin d'être exhaustive.

3. Structure



4. Constituants

4.1. *Elément instanciable*

Un élément du modèle qui peut être instancié à volonté. Un élément instanciable fournit un « prototype » pour créer des instances personnalisées.

4.2. *Propriété*

Une partie formelle constitutive d'un élément instanciable. Une propriété n'a pas de valeur ou de contenu en elle-même. Différentes valeurs peuvent lui être associées au niveau de chaque instance.

4.3. *Instance*

Un élément conforme à un élément instanciable donné. Une instance possède un ensemble de liaisons.

4.4. *Liaison*

Une liaison met en correspondance (connecte ou affecte) une propriété avec une valeur effective dans le contexte d'une instance donnée.

4.5. *Valeur*

Un élément attaché à une propriété dans le contexte d'une instance donnée. La sémantique d'un tel élément est fortement dépendante du domaine métier. C'est pourquoi le terme « valeur » utilisé ici peut être inapproprié dans certaines situations.

5. Exemples

5.1. *Déclaration et appels de fonctions en C*

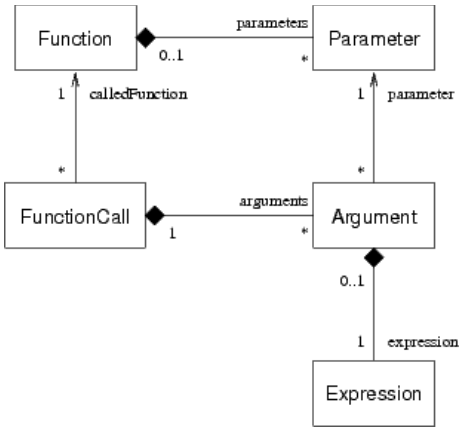
Dans les langages procéduraux, les fragments de code réutilisables sont placés dans des sous-programmes. Un sous-programme possède généralement un nom et une liste de paramètres formels. Pour appeler un sous-programme, il faut fournir les informations suivantes :

- une indication du sous-programme à appeler (dans les langages textuels, on donne le nom du sous-programme) ;
- une liste d'arguments. Cela consiste en fait à mettre en correspondance chaque paramètre formel du sous-programme avec une valeur (souvent présentée sous la forme d'une expression).

Nous donnons ci-dessous (à gauche) un exemple de déclaration et d'appel de fonction en langage C. Dans un métamodèle du langage C, on pourra trouver le motif

suivant (figure de droite). Cet extrait illustre le fait que dans un appel de fonction, chaque argument associe un paramètre de la fonction et une expression :

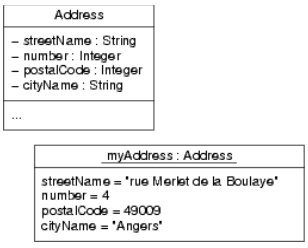
```
int foo(int a, int b);

int main() {
   ...
   z = foo(x+y, x-y);
}
```

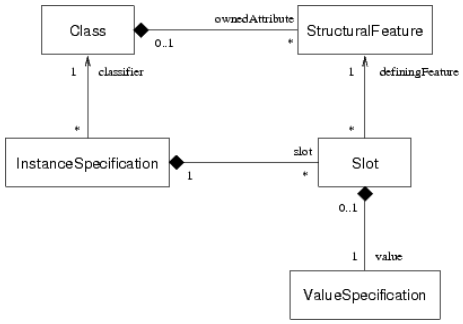


5.2. *Instanciation de classes en UML*

Le package *kernel* de la spécification UML2 inclut les concepts de classe et d'instance. Si l'on simplifie quelque peu la spécification, on peut extraire les informations suivantes :

- une classe possède des attributs ;
- une instance est un objet conforme à une classe donnée ;
- une instance possède, pour chaque attribut, un slot auquel est affectée une valeur.



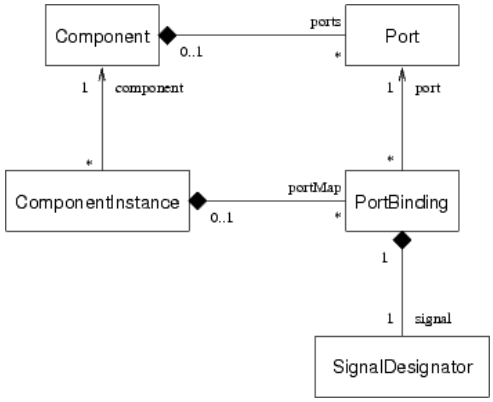


Les figures ci-dessus représentent : à gauche un exemple de modèle UML comprenant une classe et une instance; à droite un extrait simplifié du métamodèle UML2.

5.3. *Instanciations de composants en VHDL*

VHDL est un langage de description de matériel qui fournit, entre autres, les concepts de composant et d'instance de composant. Une description de composant (introduite par les mots-clés *entity* ou *component*) ne présente que l'interface externe d'un fragment de matériel réutilisable : on donne notamment la liste de ses ports d'entrée/sortie. Lors d'une instanciation de composant, la clause *port map* permet de connecter chaque port à un signal existant. Dans l'exemple ci-dessous (à gauche), le composant *Foo* est déclaré et un instance *U* de ce composant est créée. Le port a de cette instance est relié aux 8 bits de poids fort du signal *x* ; le port *b* n'est pas connecté ; le port *s* est connecté au signal *y*.

```
component Foo
  port (
    a : in bit_vector(7 downto 0);
    b : in bit_vector(7 downto 0);
    s : out bit
  );
end component;

signal x : bit_vector(15 downto 0);
signal y : bit;
...
U : Foo port map (
    a => x(15 downto 8),
    b => open,
    s => y
  );
```



On donne ci-dessus, à droite, un extrait simplifié du métamodèle VHDL illustrant les notions de déclaration et d'instanciation de composant.

12. Bibliographie

Object Management Group – *Unified Modeling Language: Superstructure, version 2.0* – August 2005

Object Management Group – *Meta-Object Facility (MOF) Core Specification, version 2.0* – January 2006

IEEE/DASC/VASG – *IEEE Standard VHDL Language Reference Manual (1076-2000)* – May 2002

Brian W. Kernighan, Dennis Ritchie – *The C Programming Language* – Prentice Hall, March 1998

Motif pour la métamodélisation

Interfaçage entre contextes

Alain Plantec — Vincent Ribaud

EA3883, LISyC, Université de Bretagne Occidentale,
C.S. 93837, 29238 Brest Cedex 3
{alain.plantec,vincent.ribaud}@univ-brest.fr

RÉSUMÉ. *Les métamodèles sont très souvent conçus à partir de la définition de contextes dans lesquels sont spécifiés les différents éléments du langage modélisé. Les langages proposent alors un ou plusieurs mécanismes d'interfaçage entre contextes permettant la réutilisation d'élements d'un contexte cible depuis un contexte source. Interfaçage entre contextes est un motif pour l'implantation de la réutilisation inter-contextes*

ABSTRACT. *Metamodels are often designed with contexts definition which include language elements definitions. Target context elements reuse from a source context are implemented using one or several interfacing mechanism. Contexts interfacing is a pattern for such implementation*

MOTS-CLÉS : *métamodèle, contexte, interfaçage, réutilisation*

KEYWORDS: *metamodel, context, interfacing, reuse*

1. Motivation

La plupart des langages de modélisation ou de programmation permettent la spécification d'éléments de modélisation au sein d'un contexte. Ce principe est à la base de la notion de visibilité des concepts. Un élément est implicitement visible dans le contexte auquel il appartient : il peut ainsi être référencé par les autres éléments de modélisation du même contexte. On parle de *paquetage* en *java*, d'*espace de nommage* en *C++* ou de *schéma* en *EXPRESS*.

Le développement modulaire, la compilation séparée et plus récemment l'ingéniérie des modèles permettent de mettre en oeuvre la réutilisation de concepts. Le principe très majoritairement retenu est d'indiquer, dans le contexte utilisateur, les concepts réutilisés. Les contextes et les concepts réutilisables sont nommés.

La déclaration des concepts utilisés s'effectue :

– soit en indiquant le nom du contexte, tous les concepts du contexte utilisé sont alors implicitement inclus ;

– soit en indiquant une liste de noms de concepts par contexte utilisé, les concepts sont alors explicitement inclus.

Un concept utilisé pouvant lui même référencer d'autres concepts, par transitivité, ces autres concepts peuvent devenir implicitement inclus.

L'interfaçage peut avoir pour conséquence d'introduire une ambiguité de référence lorsque qu'un concept importé porte le même nom qu'un concept local ou qu'un concept importé par ailleur. Suivant le langage et suivant l'environnement hôte, ce problème est principalement géré par le nommage complet du chemin incluant le nom des contextes parents ou par le renommage du concept utilisé.

2. Indication d'utilisation

Interfaçage entre contextes est de portée large puisqu'il peut s'appliquer aussi bien pour des modèles textuels que pour des modèles graphiques. Le motif permet la constitution de graphes reliant les différents contextes d'un modèle tout en autorisant la gestion des relations inter-contextes cycliques.

Ce motif est principalement utilisé pour bénéficier de la réutilisation entre différents contextes. La notion d'*interfaçage* permet à un environnement de localiser les concepts effectivement utilisés et ainsi de distinguer les concepts locaux des concepts externes.

Pour la compilation, la génération de code ou pour le remodelage cette distinction et indispensable pour que le code ou le modèle synthétisé puisse refléter de l'organisation modulaire des spécifications d'entrée. Un *Interfaçage entre contextes* est utilisé par un compilateur pour construire la table des symboles exploitée par l'éditeur de liens. Un générateur de code exploite un *Interfaçage entre contextes* pour implanter

les imports et organiser le code généré. Pour la rétro ingéniérie, l'organisation modulaire d'un programme peut se traduire directement dans l'environnement de modélisation. Les graphes obtenus peuvent être analysés (arbres des dépendances, références croisées ...) et réorganisés.

Le motif est utile pour la vérification des modèles et l'élaboration de messages d'erreur intelligibles. En effet, la réutilisation ainsi représentée spécifie une intention de réutilisation. A l'issue d'un processus de résolution, l'intention peut se révéler insoluble. Les métadonnées disponibles permettent alors de disposer de suffisamment d'informations pour construire des rapports d'erreur intelligibles.

Dans un environnement, l'aide à la saisie peut tirer partie de ce motif pour proposer les symboles par modules. Seul les symboles externes réellement importés sont proposés en tenant compte du renommage éventuel.

3. Structure

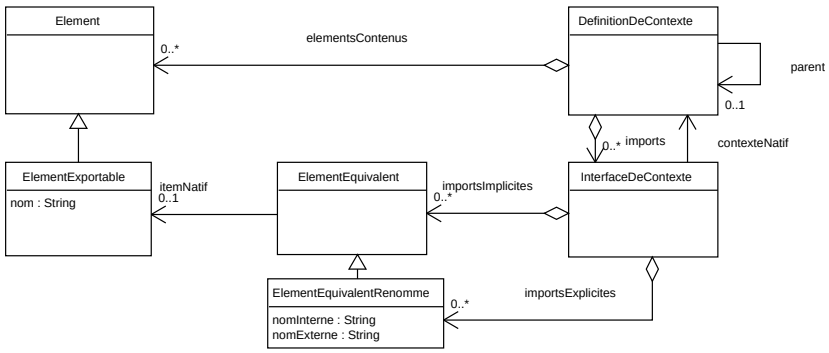


Figure 1. *Structure du motif "Interfaçage entre contextes"*

4. Constituants

4.1. *Element*

Représente tout élément de modèle spécifié dans un contexte.

4.2. *ElementExportable*

Représente un *Element* qui peut être exporté, soit, réutilisé par un contexte source. Très généralement, la déclaration d'un import repose sur un identifiant (*nom*), constituant une clé non ambiguë d'accès à l'élément dans le contexte auquel il appartient.

4.3. *DefinitionDeContexte*

Un *DefinitionDeContexte* constitue le contexte des éléments modélisés (*elements-Contenus*). Un *DefinitionDeContexte* peut être éventuellement lui même déclaré dans un *DefinitionDeContexte* parent.

4.4. *InterfaceDeContexte*

Une *InterfaceDeContexte* permet d'accéder aux éléments importés (*imports*) depuis un contexte source. Elle référence un contexte cible (*contextNatif*) dont tout ou partie des *ElementsExportable* sont réutilisés par le contexte source.

4.5. *ElementEquivalent*

Un *ElementEquivalent* représente une intention de réutilisation implicite d'un *ElementsExportable*. Une réutilisation implicite étant calculée, elle est résolue par construction. Via un *ElementEquivalent*, il est donc toujours possible à l'*InterfaceDeContexte* d'accéder à l'*ElementExportable* réutilisé.

La création d'un *ElementEquivalent* est effectué lorsqu'un *DefinitionDeContexte* source spécifie la réutilisation globale de tous les *ElementExportable* d'un *Definition-DeContexte* cible ou par le traitement des imports implicites induits par transitivité.

4.6. *ElementEquivalentRenomme*

Un *ElementEquivalentRenomme* représente une intention de réutilisation explicite d'un *ElementExportable*. *nomExterne* constitue la clé qui identifie l'*ElementExportable* dans le *DefinitionDeContexte* cible. *nomInterne* représente la clé qui identifie l'*ElementExportable* dans le *DefinitionDeContexte* source.

L'absence de renommage peut être indiqué soit par l'absence de *nomInterne*, soit par l'égalité des deux clés *nomExterne* et *nomInterne*.

Un *ElementEquivalentRenomme* peut ne pas être résolu dans le *DefinitionDeContexte* cible. Dans ce cas, un rapport d'erreur peut utiliser la clé *nomExterne*.

5. Exemple

L'exemple de la figure 2 présente un métamodèle simplifié du langage de modélisation EXPRESS (ISO 1994b) du standard ISO STEP (ISO 1994a) :

– le contexte (*context_definition*) comprend la définition de types (*named_type*), d'entités (*entity_definition*), de procedures et de functions (*statements_container*) ;

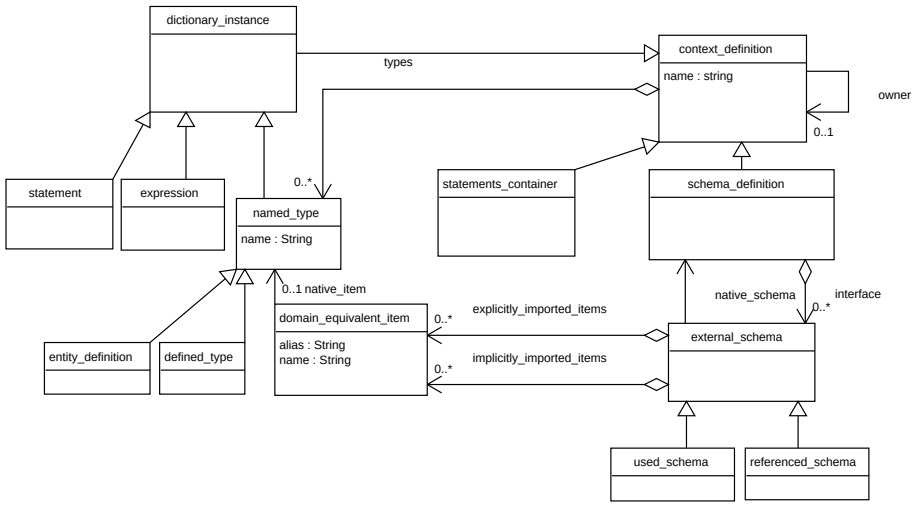


Figure 2. *Extrait d'un métamodèle simplifié d'EXPRESS*

– le schéma (*schema_definition*) est le contexte racine au niveau duquel il est possible d'importer les définitions d'autres schémas et joue donc le rôle de *DefinitionDeContexte*.

– *external_schema* joue le rôle de *InterfaceDeContexte* ; EXPRESS dispose de deux mécanismes de réulisation représentés par *used_schema* et *referenced_schema* qui spécialisent *external_schema* ;

– l'intention d'une réutilisation représenté par *domain_equivalent_item* peut être précisé avec un alias pour indiquer un renommage ; Dans cet exemple, *domain_equivalent_item* joue à la fois le rôle de *ElementEquivalent* et *ElementEquivalentRenomme*.

– la réutilisation est implicite ou explicite.

La norme précise l'algorithme de calcul des réutilisations implicites et de résolution des références. Ces algorithmes s'appuient sur les *external_schema* qui doivent dans un premier temps être eux même résolus.

Les procédures et les fonctions (les *statements_container*) d'un schéma peuvent aussi être réutilisés. Cette possibilité n'apparaît pas dans le métamodèle simplifié.

6. Bibliographie

ISO, *Part 1 : Overview and fundamental principles*. 1994a.

ISO, *Part 11 : EXPRESS Language Reference Manual*. 1994b.

Actes de l'atelier de travail
Motifs de méta-modélisation

Motif pour la métamodélisation

Plateforme d'exécution

Frédéric Thomas, Safouan Taha, Ansgar Radermacher, Sébastien Gérard

DRT/DTSI/SOL/LLSP – équipe ACCORD|$_{UML}$
CEA Saclay, 91191 GIF/YVETTE CEDEX
{ frederic.thomas, safouan.taha, ansgar.radermacher, sebastien.gerard}@cea.fr

RÉSUMÉ. *Le motif pour la métamodélisation de plateforme d'exécution est un modèle de structure générique qui réunit les principaux concepts nécessaires à la description de plateforme cible.*

ABSTRACT. *The modeling pattern for execution platform is a generic structural model, which introduce basic concepts for the design of any target platform.*

MOTS-CLÉS : *modèle, métamodèle, plateforme d'exécution, ressource, service*

1. Motivation

Une des réponses aux problèmes croissants de complexité, d'hétérogénéité et d'évolutivité des systèmes est bien souvent la séparation des règles métiers d'une application vis-à-vis de son implémentation sur une plateforme cible. Le concept de plateforme est souvent essentiel et central dans les approches dirigées par les modèles. Nous pouvons par exemple citer l'approche MDA[1] proposée par l'OMG[2] [1]. Un de ses objectifs est de pouvoir, à partir d'une description indépendante de toute plateforme (PIM : Platform Independent Model) générer des applicatifs dédiés (PSM : Platform Specific Model) et ceci en s'appuyant sur une description de la plateforme d'exécution (PDM : Platform Description Model). L'objet de cet article est donc de présenter un patron générique visant à la modélisation de plateformes d'exécution.

Nous retrouvons aujourd'hui au sein de la littérature liée à l'ingénierie dirigée par les modèles, un grand nombre de définitions du concept de plateforme d'exécution. Celles-ci sont souvent liées à un domaine d'application ou à une approche de conception. Pour exemple dans [2], R. Marvie identifie les caractéristiques génériques d'une plateforme d'exécution, de même que [3] et [4] qui

1. Acronyme pour « Model Driven Architecture ».
2. Acronyme pour « Object Management Group ».

en proposent à la fois une description et une méthodologie de modélisation. Notre objectif n'est justement pas de rendre compte d'une méthodologie mais bien plus de refléter des pratiques de modélisation.

Nous conviendrons qu'une plateforme d'exécution est une plateforme qui supporte entièrement l'exécution de l'application en proposant un environnement d'exécution complet en terme de ressources (i.e. entités structurelles) et de services (i.e. entités fonctionnelles). Elle réalise de ce fait une interface suffisante aux besoins de l'application.

2. Indications d'utilisation

Notre patron de modélisation propose une structure générique de la plateforme d'exécution. Il peut être spécialisé à un niveau méta pour refléter un domaine donné. Nous le construisons à un haut niveau d'abstraction à partir duquel l'utilisateur peut spécialiser ses propres concepts métiers. Ainsi, il pourra être utilisé aussi bien pour la modélisation de plateformes matérielles que logicielles.

3. Structure

Nous considérons une plateforme d'exécution comme étant un ensemble de ressources, chaque ressource propose au moins un service et peut en nécessiter d'autres. Du point de vue de l'application, une plateforme est considérée comme étant une couche d'abstraction réalisant une ou plusieurs interfaces d'utilisation.

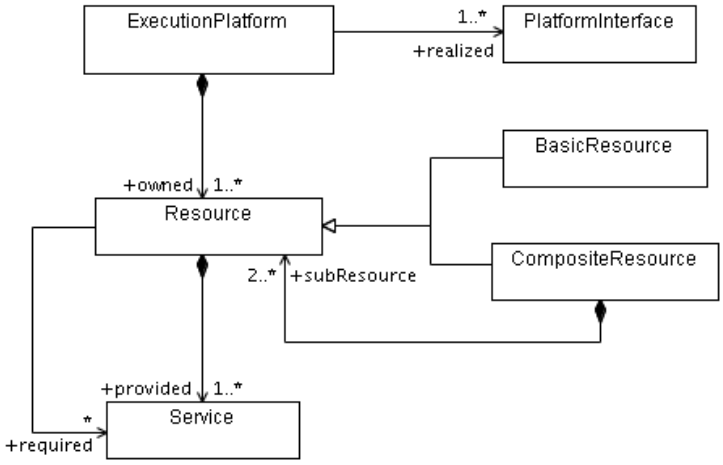


Figure 1. *Structure du métamodèle « Plateforme d'exécution »*

3. Constituants

3.1. *ExecutionPlatform*

La métaclasse *ExecutionPlatform* est un conteneur. Elle regroupe au minimum une ressource, et elle réalise une ou plusieurs *PlatformInterface*. Le rôle de *ExecutionPlatform* est de faire correspondre chaque appel de *PlatformInterface* une collaboration adéquate de ressources.

3.2. *PlatformInterface*

PlatformInterface est une métaclasse externe à la plateforme. Elle représente le contrat qui relie la plateforme d'exécution à l'application. Une telle interface peut être particulière à la plateforme modélisée ou commune à plusieurs plateformes (i.e. la norme POSIX [5] pour les systèmes d'exploitation).

3.3. *Resource*

La métaclasse *Resource* représente une entité structurelle de la plateforme d'exécution. Elle offre des services et peut nécessiter les services d'autres ressources de la même plateforme. On distingue intuitivement une ressource élémentaire, *BasicResource* d'une hiérarchique, *CompositeResource*. Cette dernière encapsule au minimum deux ressources. Une telle classification permet de décrire la structure de la plateforme à plusieurs niveaux de détail, en effet une *BasicResource* peut être raffinée en une *CompositeResource*.

3.4. *Service*

Cette métaclasse feuille représente une fonctionnalité proposée par une ressource.

4. Exemple

Nous avons appliqué le motif (Figure 1) pour concevoir un métamodèle de plateforme logicielle temps réel, nous vous en présentons un fragment (Figure 2).

Dans cet exemple, nous modélisons une plateforme possédant des ressources logicielles basiques telles que des tâches et des alarmes. Ces concepts proposent des services comme l'activation et la terminaison pour une tâche ainsi que le paramétrage pour l'alarme. Ils peuvent également nécessiter les services d'autres ressources. Par exemple une tâche périodique a besoin des services de l'horloge système.

L' « *Application Programming Interface* » (API) est l'unique représentation de la plateforme que l'utilisateur manipule. Pour exemple, une des APIs souvent réalisée par les systèmes d'exploitation temps réel est la norme POSIX [5].

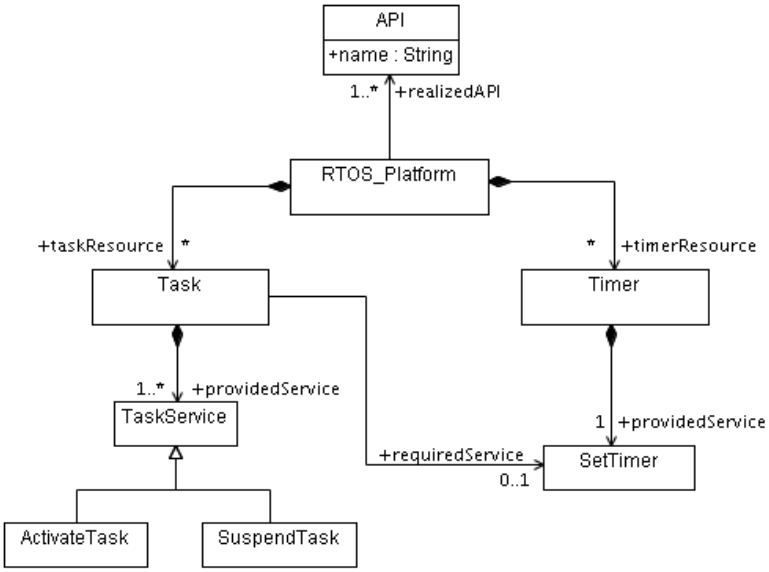


Figure 2. *Fragment d'un métamodèle d'une plateforme logicielle temps réel*

5. Conclusion

Pour conclure, l'exemple illustre une certaine simplicité d'utilisation de notre motif. Nous allons également appliquer ce motif sur un métamodèle de plateforme d'exécution matérielle. Ceci permettra d'expérimenter d'avantage la cohérence et la consistance de notre proposition.

6. Bibliographie

[1] OMG, « Model Driven Architecture », v 1.0.1, juin 2003

[2] Raphaël Marvie, Laurence Duchien et Mireille Blay-Fornarino, « Plateformes d'exécution et l'IDM », CNRS, Janvier 2005.

[3] Alberto Sangiovanni Vincentelli, «Defining Platform-based Design», EDesign of EETimes, February 2002.

[4] J.P. Andrade Almeida, R.M. Dijkman, M.J. van Sinderen, and L. Ferreira Pires. «Platform-independent modelling in MDA supporting abstract platforms», *Model Driven Architecture: Foundations and Applications (MDAFA)*, Lecture Notes in Computer Science (LNCS) 3599, Springer, pp. 174-188, 2005.

[5] «Standard for Information Technology - Portable Operating System Interface», IEEE, 2001.

Des méta-modèles au banc d'essai des patrons de conception

(Ré)utilisation et Intégration

Cédric Bouhours, Hervé Leblanc

IRIT – équipe MACAO (Modèles, Aspects et Composants pour des Architectures à Objets)
Université Paul Sabatier
118 Route de Narbonne
F-31062 TOULOUSE CEDEX 9
{bouhours,leblanc}@irit.fr

RÉSUMÉ. *Les méta-modèles jouent un double rôle. D'une part, ils structurent les modèles et ont un rôle conceptuel, et d'autre part, ils servent d'accès à l'information via des langages de requêtes et de transformations. Dans ce contexte, nous relatons une expérience faite sur deux méta-modèles, SPEM et le paquetage core de UML1.4, en y intégrant des patrons de conception, devenant de fait, candidats au statut de patron de méta-modèle.*

ABSTRACT. *Meta-models act a dual role. First, they structure models and have a conceptual role, on the other hand, they are used to access information via query languages and transformations. So, we report an experiment on two meta-models, SPEM and core of UML1.4, which integrate design patterns, becoming indeed, pretender to status of meta-model pattern.*

MOTS-CLÉS : *patrons de conception, méta-modèles, SPEM, UML 1.4.*

KEYWORDS: *design patterns, meta-model, SPEM, UML 1.4.*

1. Un ensemble de règles OCL dédié aux patrons structurels

Les patrons de conception, tout comme les autres types de motifs, représentent un savoir-faire sous forme de microarchitecture de classes réutilisables. Les patrons de conception structurels permettent une conception simple en termes de structure (gestion des liens d'héritage et d'association) et élégante en termes de responsabilité entre classes. Afin de favoriser leur utilisation dans des conceptions existantes, nous nous sommes attachés à déterminer les conditions d'applications de chacun des patrons structurels proposés par (Gamma *et al.,* 1995). Pour cela, nous avons du trouver un ensemble de particularités structurelles qui, une fois repérées dans un modèle, ciblerait le fragment de modèle à transformer par l'injection d'un patron.

Un modèle alternatif à un patron est un modèle qui résout le même problème que le patron, mais dont la structure n'intègre pas la microarchitecture du patron. Il s'agit donc d'un modèle candidat à la substitution par un patron.

Pour obtenir ces alternatives, une expérience a été réalisée sur deux promotions d'élèves en cursus informatique, qui n'avaient à priori aucune connaissance sur ces patrons. Ils devaient modéliser, dans la notation UML, une série de sept problèmes solubles avec les sept patrons structurels. Sur les trois cent modèles obtenus, nous en avons sélectionné quarante qui constituaient des alternatives aux patrons, et après avoir retiré tous les doublons, il en est resté onze qui présentaient des différences structurelles significatives. Chaque modélisation obtenue (entre deux et cinq par patron) constituait donc une alternative plausible à un patron.

Chaque patron de conception structurel se caractérise par un rôle attribué à chacune des classes qui le compose, chaque rôle ayant une particularité structurelle propre. Chaque modèle alternatif se caractérise de la même façon qu'un patron. A chaque rôle du modèle alternatif est associé un ensemble de particularités structurelles dites remarquables. Celles-ci permettent de repérer dans un modèle, les classes pouvant jouer les différents rôles du patron. En associant, à chaque classe des modèles alternatifs, les rôles des patrons correspondant, et en étudiant leur structure (sous-arbres équivalents dans des hiérarchies d'héritages, répartition quantitative des associations et des compositions…), nous avons pu déduire ces particularités. Pour chaque modèle alternatif, il y a donc un ensemble de particularités remarquables associées à chacun des rôles du patron. Ces particularités sont pour le moment uniquement structurelles, la notion d'équivalence des interfaces des classes n'étant pas pour le moment dans la portée de ces travaux (Bouhours, 2006).

Ensuite, nous avons établi une méthode générique de recherche qui repère, dans le modèle à analyser, les classes pouvant jouer le rôle de référence sur le patron à identifier. Puis, en utilisant ces classes, la méthode tente d'affecter les autres rôles du patron aux autres classes du modèle liées à celles-ci. Enfin, nous avons automatisé cette détection à l'aide de règles OCL supportées par la plate-forme Neptune (Neptune, 2003).

2. Expérimentation sur des méta-modèles

Les méta-modèles UML étant instances du MOF, ils sont décrits par des diagrammes de classes, ce qui nous a permis d'y appliquer nos règles OCL. Nous les avons considérés alors comme des modèles aptes à l'intégration de patrons de conception. De ce fait, certains d'entre eux pourraient être considérés comme des patrons de méta-modèle.

D'après la spécification du langage OCL, une règle permet d'ajouter des contraintes à un diagramme UML, et de fait, d'affiner la sémantique de modèles ou de méta-modèles, comme proposé par *the precise UML group* quant aux méta-

modèles UML. Pour notre part, nous avons détourné OCL de son but initial pour rechercher des motifs structurels sur des modèles, en naviguant sur les liens du méta-modèle. Normalement, une règle renvoie un booléen qui indique si un diagramme respecte ou non la règle concernée. Dans notre cas, nous souhaitions que les règles OCL retournent des collections de classes correspondant aux différents rôles de chaque patron. La plate-forme Neptune ayant la particularité de retourner le contexte d'une erreur (ensemble des éléments du méta-modèle sur lequel était appliquée la règle), nous avons codé nos règles OCL de manière à ce qu'elles retournent toujours faux, et que le contexte de l'erreur corresponde à l'ensemble des classes qui nous intéressait.

Le résultat se présente sous la forme de plusieurs collections de classes, chacune correspondant à un rôle du patron. Par exemple, le résultat de l'application des règles concernant le patron *Composite* se présente sous la forme de trois ensembles. Ils contiennent respectivement les classes pouvant jouer les rôles de « Composant », « Composite » et « Feuille ».

Sur les sept patrons structurels du GOF, nous en avons éliminé quatre :

– Façade, car les liens de dépendances entre les paquetages des méta-modèles de l'OMG sont en général matérialisés par des liens d'héritages, suggérant une architecture logicielle en couches.

– Pont, car les liaisons entre abstraction et implémentation n'ont pas à être définies à ce niveau de modélisation.

– Procuration et Poids mouche, car ils ne sont pas détectables structurellement. Les problèmes qu'ils résolvent émanent d'exigences non fonctionnelles.

L'application des règles concernant les trois patrons restants Composite, Décorateur et Adaptateur, nous a permis de cibler des fragments de méta-modèles à partir desquels nous proposons la discussion suivante. Notre expérimentation a porté sur le paquetage core de UML1.4 et sur le méta-modèle SPEM, pour éviter l'explosion combinatoire des possibilités d'intégration de patrons, et pour faciliter la vérification de notre hypothèse.

3. Composite et Décorateur intégrés dans le SPEM

Le SPEM (Software Process Engineering Metamodel) est un méta-modèle spécifique aux processus de développement proposé par l'OMG (OMG, 2005). Le noyau se compose en particulier des classes WorkDefinition (découpage du processus en activité élémentaire, itération, phase et cycle de vie), ProcessRole (entité reliant des compétences requises à des activités) et WorkProduct (paramètre en entrée-sortie des activités). La figure 1 montre le fragment ciblé par nos règles. Elles font apparaître deux patrons de conception susceptibles d'être intégrés : le patron Composite (relation réflexive subwork sur la super classe WorkDefinition) et le patron Décorateur sur le même ensemble de classes.

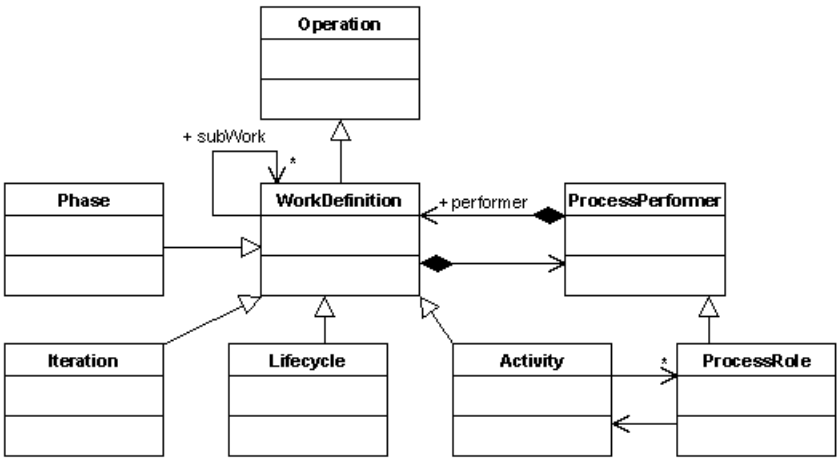


Figure 1. *Fragment du SPEM 1.1*

Une analyse plus fine nous permet de constater que la classe WorkDefinition joue plusieurs rôles : les rôles « Composite » et « Composant » du patron Composite, et le rôle « Décorateur » du patron Décorateur. La classe Activity ne pouvant se décomposer qu'en travaux atomiques nommés Step, elle joue nécessairement le rôle « Feuille » du patron Composite. Les classes Lifecycle, Phase et Iteration peuvent jouer le rôle « Décorateurs concrets » : Phase et Iteration ajoutent les notions de temps et de buts à atteindre, Lifecycle spécialise la relation subWork aux instances de Phase et relie celles-ci aux processus et composants de processus considérés comme des référentiels d'activités.

La figure 2 montre le fragment du méta-modèle SPEM, enrichie d'une intégration explicite des deux patrons Composite et Décorateur. En intégrant la classe WorkComponent, jouant explicitement le rôle de « Composant », nous simplifions la définition de la classe WorkDefinition jouant uniquement les rôles « Composite » et « Décorateur ».

L'intégration de ces deux patrons de conception a deux effets : d'une part elle clarifie les possibilités de chacune des classes de ce méta-modèle, en leur attribuant des rôles précis. D'autre part, elle permet de préciser la sémantique du méta-modèle sans ajouter de *well formedness rules* supplémentaires ou de descriptions en langage naturel. Par exemple, le rôle « Feuille » de la classe Activity permet de se passer des précisions suivantes (OMG,§7-3,p7-5,2005) : « *Although this is not explicity prohibited, an Activity does not normally use the subWork structure inherited from WorkDefinition; instead decomposition within Activity is done using Steps* ». En revanche, des contraintes de composabilité entre les différents décorateurs concrets restent nécessaires.

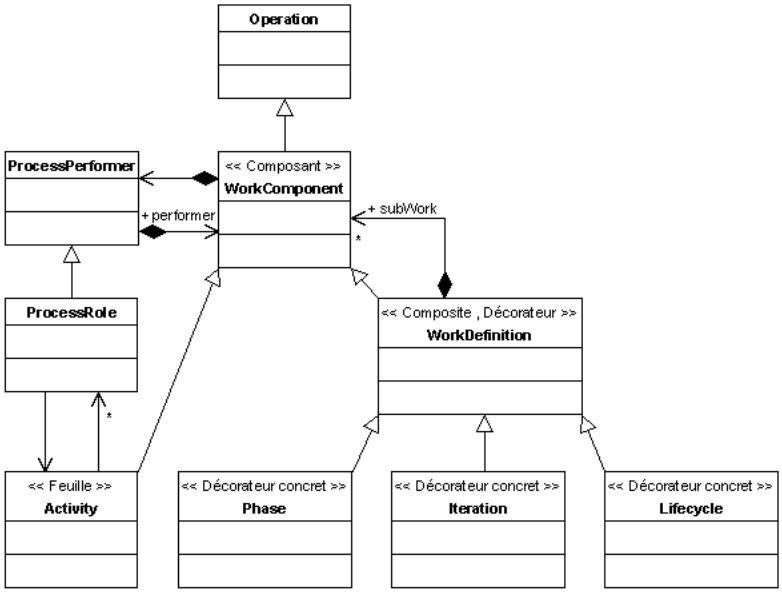


Figure 2. *Intégration des patrons Composite et Décorateur dans le SPEM 1.1*

4. Composite et Adaptateur pour simplification du paquetage core de UML 1.4

Lors de l'application des règles OCL sur le noyau du méta-modèle UML 1.4, nous obtenons un patron Composite sur les classes Namespace (rôle « Composite ») et ModelElement (rôle « Composant »). A priori, la classe GeneralizableElement est « Feuille », mais pour des raisons conceptuelles, les classes Classifier et Package sont en même temps « Feuille » (hérite de GeneralizableElement) et « Composite » (hérite de Namespace). Il existe donc une double structure d'héritage multiple et répété intégrée dans un patron Composite.

Nous proposons alors une simplification de conception en intégrant le patron Adaptateur, chargé d'implémenter l'interface de GeneralizableElement pour l'ensemble de ses anciennes sous classes, comme cela est suggéré à la figure 3.

5. Conclusion

Cette détection, prévue au départ pour des modèles UML, nous a permis de détecter des patrons dit de conception sur des méta-modèles de l'OMG. Nous considérons que les patrons Composite et Décorateur sont des candidats au statut de patrons de méta-modèle.

Afin d'étendre la discussion, il nous semblerait intéressant d'étudier le statut des autres patrons de conception. Si nous éliminons les patrons créateurs, chargés de gérer l'instanciation d'objets complexes (agrégats, frameworks), il subsiste certains patrons comportementaux candidats : le patron Interprète, servant de support au méta-modèle dédié aux langages spécifiques de domaines (DSL), le patron Chaîne de responsabilité, chargé d'expliciter les liens de navigation au travers des classes du méta-modèle et le patron Médiateur simplifiant la navigation dans les modèles. De plus, le patron structurel Pont peut être réutilisé pour la séparation propre entre les éléments de modélisation et les représentations graphiques associées. Pour terminer, deux questions nous semblent intéressantes à débattre. Les patrons métier sont-ils candidats à être des patrons de méta-modèles dédiés ? Les patrons de méta-modèles peuvent-ils être structurés à l'image des patrons composite (Riehle, 1997) ?

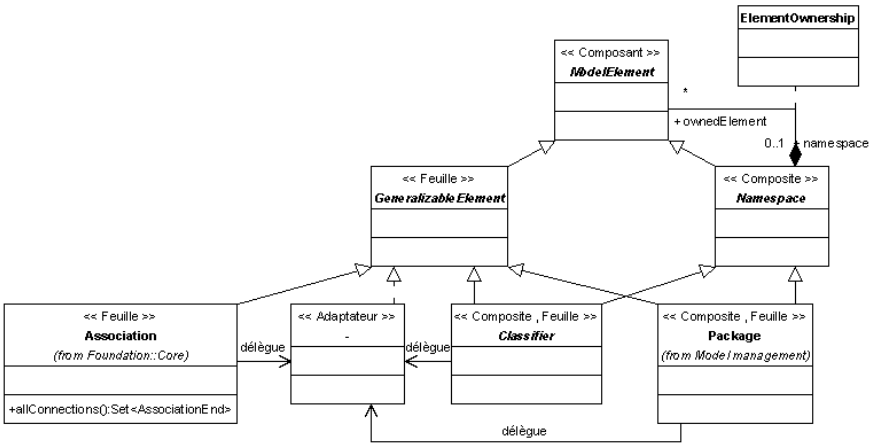


Figure 3. *Intégration d'un Adaptateur dans un fragment du noyau de UML 1.4*

6. Bibliographie

Bouhours C., Détection de particularités structurelles de modèles incitant à l'injection de patrons de conception, Master de recherche, Université Paul Sabatier, Toulouse, 2006.

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional, 1995.

Neptune, Nice Environment with a Process and Tools using Norms - UML, XML and XMI - and Example, [w] http://neptune.irit.fr, 2003.

Object Management Group, Software Process Engineering Metamodel Specification, version 1.1 formal/05-01-06, 2005.

Riehle D. « Composite Design Patterns », *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications OOPSLA '97*. ACM Press, 1997. Page 218-228.